Universität Trier

Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK)



Trier, den 01.04.2015

B. Baltes-Götz

R-Pakete und -Syntax in SPSS Statistics verwenden

Inhaltsübersicht

Herausgeber: Universität Trier
Zentrum für Informations-, Medien und Kommunikationstechnologie (ZIMK)
Universitätsring 15
D-54286 Trier
Tel.: (0651) 201-3417, Fax.: (0651) 3921
Autor: Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)

Vorwort

In diesem Manuskript geht es um die Nutzung der freien Statistik-Entwicklungsumgebung R als Erweiterung zu SPSS Statistics. Wer die Benutzerfreundlichkeit und Funktionsvielfalt von SPSS Statistics schätzt, aber auch in R realisierte Lösungen nutzen und vielleicht sogar eigene Lösungen in R entwickeln möchte, kann sich über die Kooperationsbereitschaft der beiden Programme freuen und hat ein leistungsfähiges Gespann zur Verfügung. Im Manuskript werden SPSS Statistics 22 und R 2.15 verwendet.

Die aktuelle Version des Manuskripts ist als PDF-Dokument zusammen mit den im Kurs benutzen Dateien auf dem Webserver der Universität Trier von der Startseite (http://www.uni-trier.de/) ausgehend folgendermaßen zu finden:

ZIMK (Rechenzentrum) > Infos für Studierende > EDV-Dokumentationen > Statistik > R-Pakete und -Syntax in SPSS Statistics verwenden

Kritik und Verbesserungsvorschläge zum Manuskript werden dankbar entgegen genommen (z.B. unter der Mail-Adresse baltes@uni-trier.de).

Trier, im April 2015 Bernhard Baltes-Götz

1 Einleitung

R ist eine freie *Programmierumgebung für Datenanalyse und Grafik* (dt. Übersetzung des Untertitels von Venables et al. 2014), die als Implementation der statistik-orientierten Programmiersprache **S** (Chambers 1998) entstanden ist.

Dank einer von IBM SPSS unter dem Namen **R-Essentials** gelieferten Integrationslösung kann **R** von SPSS Statistics aus genutzt werden, z.B. ...

- zur Verwendung der zahlreichen Analysefunktionen in **R**-Paketen
  So werden Analyseoptionen zugänglich, die in SPSS fehlen, z.B. robuste Regression, polychorische Korrelation, Rasch-Itemanalyse
- zur Realisation von Algorithmen in der Programmiersprache **R**

Dabei ist es problemlos möglich, Variablen aus einem SPSS-Datenblatt in einen **R** - Data Frame zu kopieren und in **R**-Funktionen über ihre SPSS-Namen anzusprechen.

Die mit **R** erzeugten Ergebnisse können wiederum an SPSS übergeben werden:

- Ausgaben
  Normale **R**-Ausgaben landen als Text im SPSS-Ausgabefenster. Durch Verwendung von Funktionen des SPSS-Erweiterungspakets für **R**, das zur Integrationslösung gehört, ist es aber auch möglich, SPSS-Pivot-Tabellen in **R** zu erstellen.
- Variablen
  Aus den mit **R** generierten oder modifizierten Variablen lässt sich ein SPSS-Datenblatt erstellen.

Um **R**-Funktionen auf SPSS-Variablen anzuwenden, erstellt man im Normalfall in einem SPSS-Syntaxfenster einen Block mit **R**-Syntax, eingerahmt durch die SPSS-Kommandos `BEGIN PROGRAM R` und `END PROGRAM`. Im folgenden Beispiel wird für die SPSS-Variablen `y` und `x` eine robuste Regressionsanalyse mit Hilfe der **R**-Funktion **rlm()** gerechnet (vgl. Abschnitt 3.1):

```
* Robuste Regression per rlm und Huber-M-Schätzer.
BEGIN PROGRAM R.
library(MASS)
casedata <- spssdata.GetDataFromSPSS()
huber <- rlm(y ~ x, data = casedata, method = "MM")
summary(huber)
END PROGRAM.
```

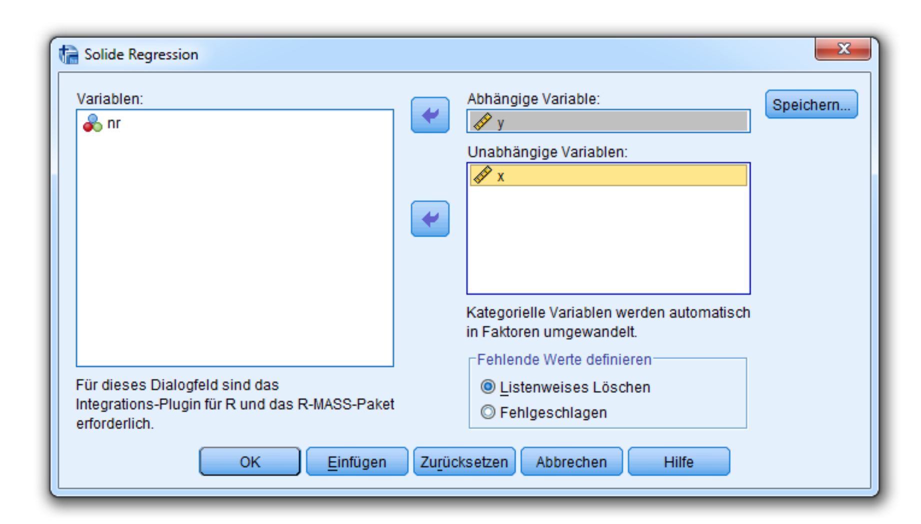
Man erhält die Ausgabe der **R**-Funktion **summary()** als Text im SPSS-Ausgabefenster (Viewer).

Im Rahmen der **R**-Integrationslösung zu SPSS Statistics ist es aber auch möglich, ein **SPSS-Erweiterungskommando** zu erstellen, das wie gewöhnliche SPSS-Syntax zu benutzen ist und dabei im Hintergrund mit **R** arbeitet. Für das obige Beispiel ist diese Arbeit von SPSS-Entwicklern erledigt worden, so dass SPSS-Anwender die robuste Regression mit **R** über vertraute Syntax anfordern können:

```
SPSSINC ROBUST REGR DEPENDENT = y  ENTER = x
  /OPTIONS MISSING=LISTWISE
  /SAVE.
```

Dass im Beispiel die **R**-Ausgaben in **SPSS-Pivot-Tabellen** gewandelt werden, steigert den Benutzerkomfort noch (siehe Abschnitt 3.1).

Alternativ oder ergänzend zu einem Erweiterungskommando kann ein **benutzerdefinierter Dialog** definiert und in das SPSS-Menü integriert werden, so dass die mit Hilfe von **R** implementierte Funktionalität auch ohne Syntaxfenster nutzbar ist, z.B.:



Dieser Dialog bildet zusammen mit dem eben vorgestellten Erweiterungskommando und der realisierenden **R**-Syntax ein **Erweiterungsbundle**, das in einer Datei mit der Namenserweiterung **SPE** angeboten wird. Es handelt sich um das zusammen mit den **R-Essentials** installierte Bundle **SPSSINC ROBUST REGR** zur Unterstützung der robusten Regression. Es sind auch benutzerdefinierte Dialoge *ohne* begleitendes Erweiterungskommando verfügbar, die in einer Datei mit der Namenserweiterung **SPD** angeboten werden. In Abschnitt 2 wird erläutert, wie die für SPSS Statistics 22 zahlreich vorhandenen Erweiterungsbundles und benutzerdefinierten Dialoge auf **R**-Basis installiert werden können.

2 SPSS-Funktionserweiterungen auf R-Basis installieren

Um die **R**-basierten Funktionserweiterungen für SPSS Statistics 22 nutzen zu können, sind die folgenden Installationen auszuführen:

- **SPSS Statistics 22**
- **R 2.15**
- **R Essentials zu SPSS Statistics 22**
  Einige Erweiterungen benötigen zusätzlich die **Python Essentials zu SPSS 22**. In den R-Essentials sind bereits etliche Erweiterungsbundles enthalten (z.B. das in der Einleitung vorgestellte Bundle SPSSINC ROBUST REGR).
- **Spezielle Erweiterungsbundles oder benutzerdefinierte Dialoge**

2.1 Python- und R-Essentials

2.1.1 Python-Essentials

Python ist eine attraktive Skriptsprache, die für Automatisierungszwecke in SPSS Statistics verwendet werden kann. Das mit den **R**-Essentials gelieferte Erweiterungskommando für heterogene Korrelationen (siehe Abschnitt 3.3) benötigt Python, so dass Sie auch die **Python-Essentials** installieren sollten. Dies kann bequem über eine Option im Installations-Assistenten von SPSS Statistics 22 geschehen. Es ist also kein separater Download erforderlich.

Wenn Sie bei der Installation von SPSS Statistics 22 die Python-Option wählen, werden die folgenden Bestandteile der Python-Essentials eingerichtet:

- Python 2.7.1
- Python-Integration-Plugin für SPSS Statistics 22
- Python-basierte Erweiterungsbundles
  Es werden etliche mit Hilfe von Python implementierte SPSS-Erweiterungskommandos samt zugehöriger benutzerdefinierter Dialoge mitgeliefert, z.B. zum automatischen Erstellen der Kodiervariablen zu einem Faktor (Menübefehl: **Transformieren > Dummy-Variablen erstellen**).

Sollten Sie bei der Installation von SPSS Statistics 22 auf die Python-Option verzichtet haben, können Sie deren Installation so nachholen:

- Installationsprogramm zu SPSS Statistics 22 erneut starten (unter Windows-64: **SPSS_Statistics_22_win64_.exe**)
- Im Dialog **Programmverwaltung** wählen: **Programm ändern**
- Im Dialog **IBM SPSS Statistics - Essentials for Python** der Installation zustimmen

2.1.2 R-Essentials

Die **R**-Essentials zu SPSS Statistics 22 werden über die folgende Webseite angeboten:

http://www.ibm.com/developerworks/spssdevcentral

Hier ist der Link Downloads for IBM® SPSS® Statistics zu wählen. Auf dem weiteren Weg zum Download besteht die Firma IBM auf einer Registrierung.

Als Ergänzung zu SPSS Statistics 22 mit FixPack 1 auf einem Windows-System mit 64-Bit – Architektur erhält man z.B. die folgende Installationsdatei:

SPSS_Statistics_REssentials_22.0-FP1_win64.exe

In dem auf derselben Webseite unter dem Titel **Installation Instructions for Windows** angebotenen Archiv

SPSS_Statistics_REssentials_Installation_Documents_22_win.zip

findet sich die folgende PDF-Datei mit Installationshinweisen:

Essentials for R Installation Instructions.pdf

In den **R**-Essentials zu SPSS Statistics 22 sind enthalten:

- Das **R**-Integration-Plugin für SPSS Statistics 22
- Das **R**-Paket **spss220**
- **R**-basierte Erweiterungsbundles
  Es werden etliche mit Hilfe von **R** implementierte Erweiterungsbundles installiert, die aus einem Erweiterungskommando und einen benutzerdefinierten Dialog bestehen (siehe Abschnitt 3).

Zusammen mit einer Version von SPSS Statistics und dem zugehörigen **R**-Integrationspaket ist eine fest vorgegeben **R**-Version zu verwenden. Die von SPSS Statistics unterstützte **R**-Version ist in der Regel nicht ganz aktuell (ca. 1 Jahr alt). Mit SPSS Statistics 22 ist die **R**-Version 2.15 zu verwenden.

Im **R** Essentials - Installationspaket zu SPSS Statistics 22 ist die vorausgesetzte **R**-Version *nicht* enthalten. Diese muss aus anderer Quelle beschafft und vor den **R**-Essential installiert werden. den. Das eben genannte PDF-Dokument empfiehlt, die **R**-Version 2.15.2 zu verwenden und nennt u.a. den folgenden Download-Link:

ftp://ftp.stat.math.ethz.ch/Software/CRAN/bin/windows/base/old/

Gehen Sie folgendermaßen vor, um SPSS Statistics 22, **R** 2.15 und die zugehörigen **R**-Essentials zu installieren:

- **SPSS Statistics 22 (inkl. Python-Essentials und dem aktuellen FixPack)**
  Falls noch nicht geschehen, müssen Sie zuerst SPSS Statistics 22 installieren, wobei Sie nicht auf die Option der **Python-Essentials** verzichten sollten (vgl. Abschnitt 2.1.1). Anschließend sollten Sie das aktuelle FixPack zu SPSS Statistics 22 installieren.
- **R 2.15.2** (oder die letzte Version **R 2.15.3** aus der 2er - Generation von **R**)
  Wenn Sie **R** auf einem Rechner mit 64-Bit – Windows installieren, müssen Sie unbedingt die 32-Bit-Version von **R** in die Installation mit aufnehmen (= Voreinstellung).
- **R-Essentials für SPSS Statistics 22 mit dem aktuellen FixPack**
  Zusammen mit einem FixPack zu SPSS Statistics werden auch die **R**-Essentials aktualisiert. Es ist also darauf zu achten, das korrekte **R**-Essentials-Installationsprogramm zu verwenden. Zu SPSS Statistics 22 mit FixPack 1 passt z.B. auf einem Windows-Rechner mit 64-Bit - Architektur das Installationsprogramm **SPSS_Statistics_REssentials_22.0-FP1_win64.exe** zu verwenden.

2.2 Erweiterungsbundles

2.2.1 Inhalt

Erweiterungsbundles ergänzen die SPSS-Funktionalität durch zusätzliche Verfahren zur Datenanalyse und/oder -verwaltung, die durch SPSS-interne Programmieroptionen (z.B. Makrotechnik, Programmiersprache MATRIX) oder externe Programmieroptionen (z.B. **R** oder Python) realisiert werden. Zur Nutzung der erweiterten Funktionalität bietet ein Bundle ein **Erweiterungskommando** und/oder einen **benutzerdefinierten Dialog**. Das in der Einleitung vorgestellte Bundle **SPSSINC ROBUST REGR** unterstützt beide Bediensysteme. Speziell bei den *nicht* von IBM SPSS entwickelten Bundles ist die Beschränkung einen benutzerdefinierten Dialog anzutreffen. So bietet z.B. Hans Grüner vom Rechenzentrum der

FU Berlin diverse benutzerdefinierte Dialoge mit **R**-Implementierung an (vgl. Abschnitt 2.3.3), die er neuerdings in Erweiterungsbundles verpackt, um die bequeme Distribution, Installation und Aktualisierung innerhalb von SPSS 22 zu ermöglichen (siehe Abschnitt 2.2.3).

2.2.2 Erstellung

Wer ein Erweiterungsbundle mit **R** erstellten möchte, muss den Implementierungscode mit **R** verfassen und die Bedienung per Syntax und/oder Dialog unterstützen. Über die Erstellung eines Erweiterungskommandos informiert das kostenlos als PDF-Datei verfügbare Buch von Levesque (2011, S. 374ff). Wer ein benutzerdefiniertes Dialogfeld anbieten möchte, findet eine Anleitung z.B. im Core-System Benutzerhandbuch (IBM SPSS 2013, Kapitel 20). Um die erstellten Dateien in ein Erweiterungsbundle zu verpacken, startet man mit dem Menübefehl

Erweiterungsbundles > Erweiterungsbundle erstellen

Es resultiert eine Erweiterungsbundle-Datei mit der Namenserweiterung **SPE**.

2.2.3 Installation

Mit den **R**-Essentials für SPSS Statistics 22 werden etliche mit Hilfe von **R** implementierte Erweiterungsbundles installiert, die auch in das Menüsystem integriert sind:[1]

- **Analysieren > Korrelation > Heterogene Korrelationen**
- **Analysieren > Regression > Quantil-Regression**
- **Analysieren > Regression > Residuums-Heteroskedastizitätstest**
- **Analysieren > Regression > Robuste Regression**
- **Analysieren > Regression > Tobit-Regression**
- **Analysieren > Skalierung > Rasch Modell**
- **Grafik > R-Boxplot**

Wer z.B. über die folgende Webseite von IBM SPSS

https://www.ibm.com/developerworks/community/files/app?lang=en#/collection/bbe88aaf-f3cd-466a-83fb-592d48eecb1c

ein Erweiterungsbundle per SPE-Datei bezogen hat und es nun installieren möchte, wählt folgenden Menübefehl:

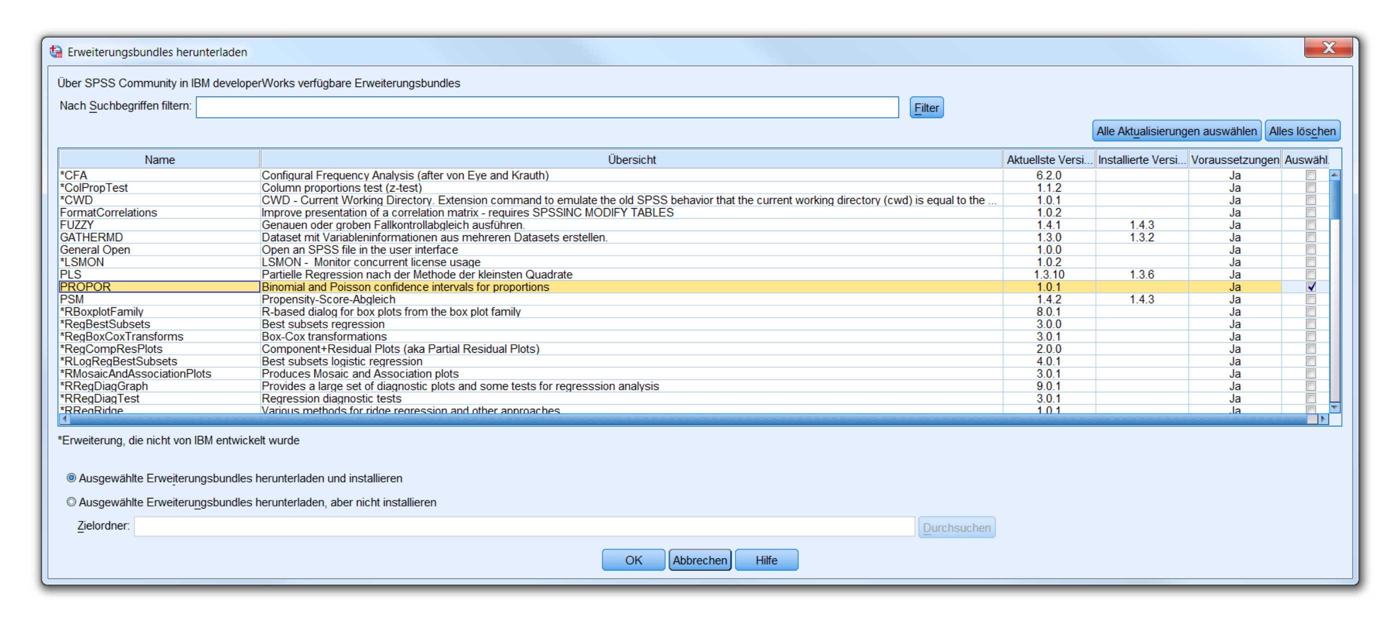
Erweiterungsbundles > Lokales Erweiterungsbundle installieren

Seit der Version 22 kann SPSS Statistics Erweiterungsbundles selbständig aus dem Internet beziehen, um sie zu installieren oder zu aktualisieren, so dass der vorherige separate Download entfällt. Nach dem folgenden Menübefehl

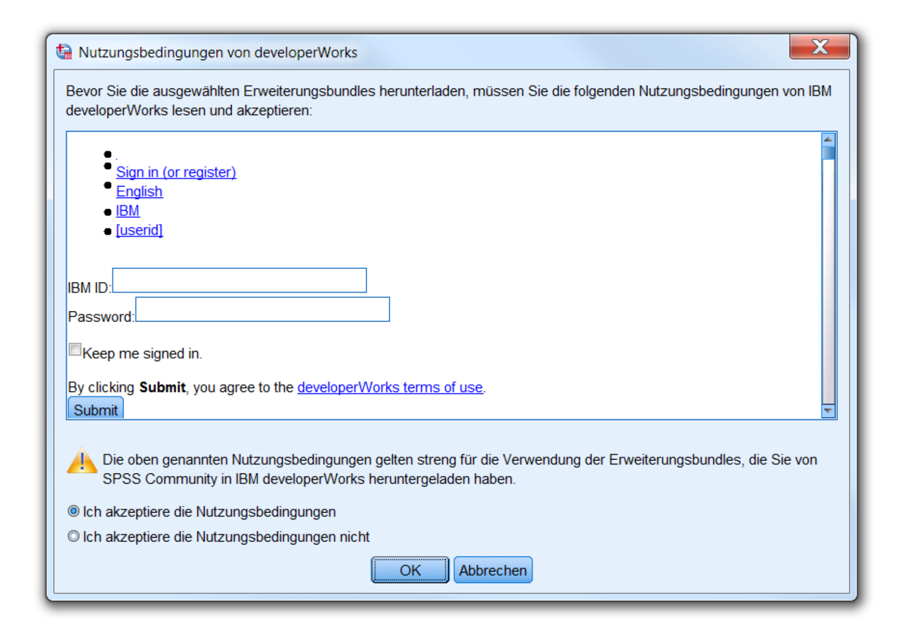
Extras > Erweiterungsbundles > Erweiterungsbundles herunterladen und installieren

erscheint ein Dialog, der die installierten und die verfügbaren Erweiterungen samt Versionsstand anzeigt:

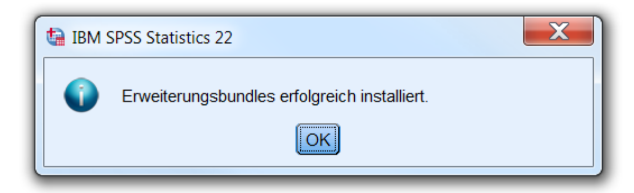
[1] Für die Erweiterung zur Berechnung von heterogenen Korrelationen sind auch die Python-Essentials erforderlich.



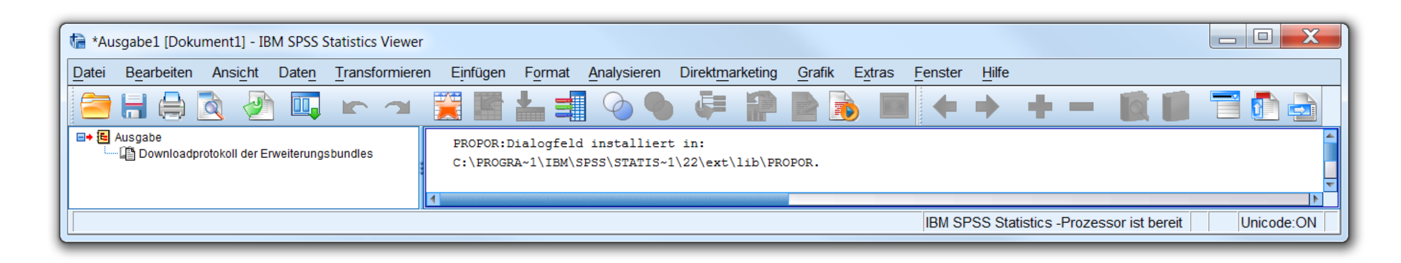
Um eine Erweiterung zu installieren oder zu aktualisieren, markiert man das **Auswählen**-Kontrollkästchen in der zugehörigen Zeile und klickt auf **OK**. Im nun erscheinenden Dialog müssen Sie lediglich die **Nutzungsbedingungen** akzeptieren. Eine **IBM ID** samt **Password** anzugeben, ist nicht erforderlich:



Nach einer erfolgreichen Installation



wird im Ausgabefenster protokolliert, wo das zum Bundle gehörige benutzerdefinierte Dialogfeld installiert worden ist, z.B.:



Wird SPSS mit Administratorrechten ausgeführt, stehen das Erweiterungskommando und das benutzerdefinierte Dialogfeld nach einem Neustart von SPSS allen Benutzern zur Verfügung. Die voreinstellten Installationsorte unter Windows 7 sind:

- Erweiterungskommandos landen zusammen mit den Dateien zur Bundle-Konfiguration in einem Unterordner von
  C:\Program Files\IBM\SPSS\Statistics\22\extensions
- Benutzerdefinierte Dialoge landen in einem Unterordner von
  C:\Program Files\IBM\SPSS\Statistics\22\ext\lib

Wird SPSS mit normalen Benutzerrechten ausgeführt, stehen das Erweiterungskommando und das Dialogfeld nach einem Neustart von SPSS dem Installateur zur Verfügung. Die voreinstellten Installationsorte unter Windows 7 sind für den Benutzer *Otto*:

- Erweiterungskommandos landen zusammen mit den Dateien zur Bundle-Konfiguration in einem Unterordner von
  C:\Users\Otto\AppData\Local\IBM\SPSS\Statistics\22\extensions
- Benutzerdefinierte Dialoge landen in einem Unterordner von
  C:\Users\Otto\AppData\Local\IBM\SPSS\Statistics\22\CustomDialogs

Auf einem Pool-PC an der Universität Trier ist die Installation eines Erweiterungsbundles mit normalen Benutzerrechten von räumlich und zeitlich begrenzter Wirkung. Zunächst landet die Installation auf einem einzelnen Pool-PC, kann sich also nicht auf andere Pool-PCs auswirken. Außerdem landet die Installation in einem Teil des Windows-Benutzerprofils, der beim Abmelden von diesem Pool-PC gelöscht wird. Zur Lösung des Problems sollten Sie vor der Installation eines Erweiterungsbundles jeweils eine benutzereigene Windows-Umgebungsvariable für Erweiterungskommandos und benutzerdefiniere Dialoge von SPSS definieren über

Systemsteuerung > Benutzerkonten > Benutzerkonten > Eigene Umgebungsvariablen ändern

Als Namen sind für diese Umgebungsvariablen sind zu verwenden:

- **SPSS_EXTENSIONS_PATH**
- **SPSS_CDIALOGS_PATH**

Zu jeder Umgebungsvariablen ist ein bereits existenter (!) Ordner anzugeben, wobei ein Ordner auf Ihrem persönlichen Laufwerk **U:** verwendet werden sollte, weil dieses Laufwerk auf jedem Pool-PC verfügbar ist, z.B.

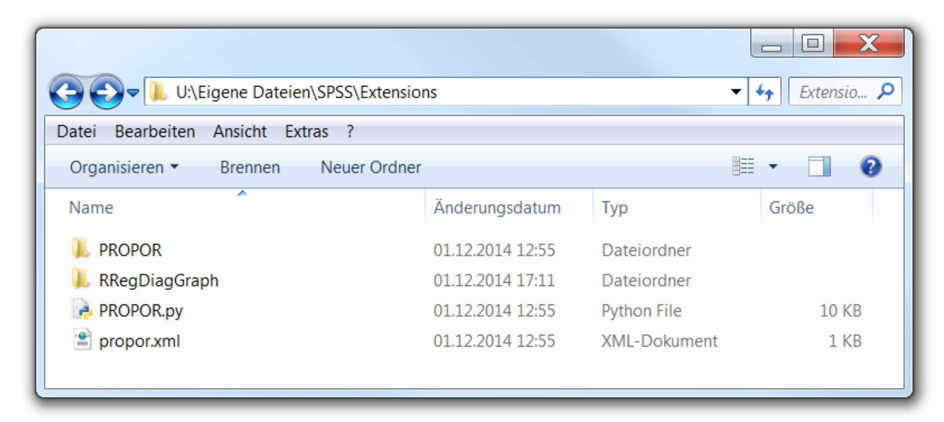
- **U:\Eigene Dateien\SPSS\Extensions**
- **U:\Eigene Dateien\SPSS\CustomDialogs**

Wenn Sie anschließend ein Erweiterungsbundle installieren, landen das Erweiterungskommando und das benutzerdefinierte Dialogfeld in den vereinbarten Ordnern und stehen auf jedem Pool-PC zur Verfügung.

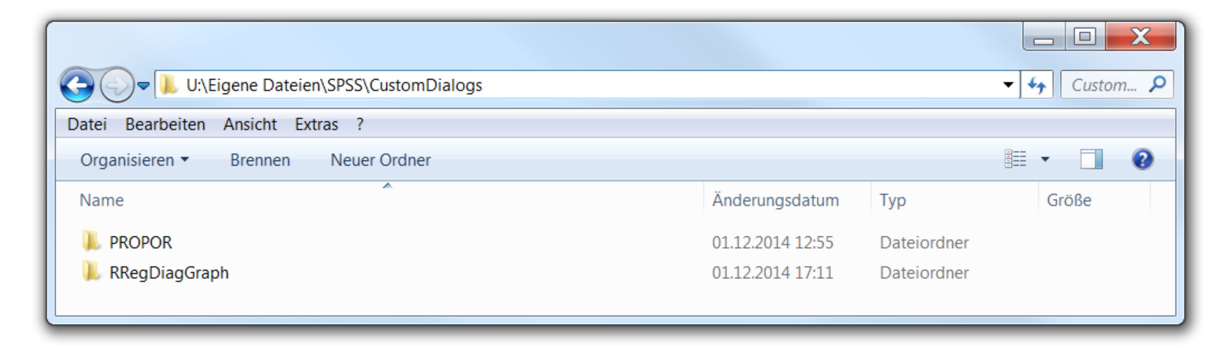
Aus der benutzerbezogenen Installation der Bundles

- **PROPOR** (mit Erweiterungskommando *und* benutzerdefiniertem Dialog)
- **RRegDiagGraph** (mit benutzerdefiniertem Dialog, ohne Erweiterungskommando)

entstanden der folgende Ordner mit den Bundle-Dateien und dem Erweiterungskommando



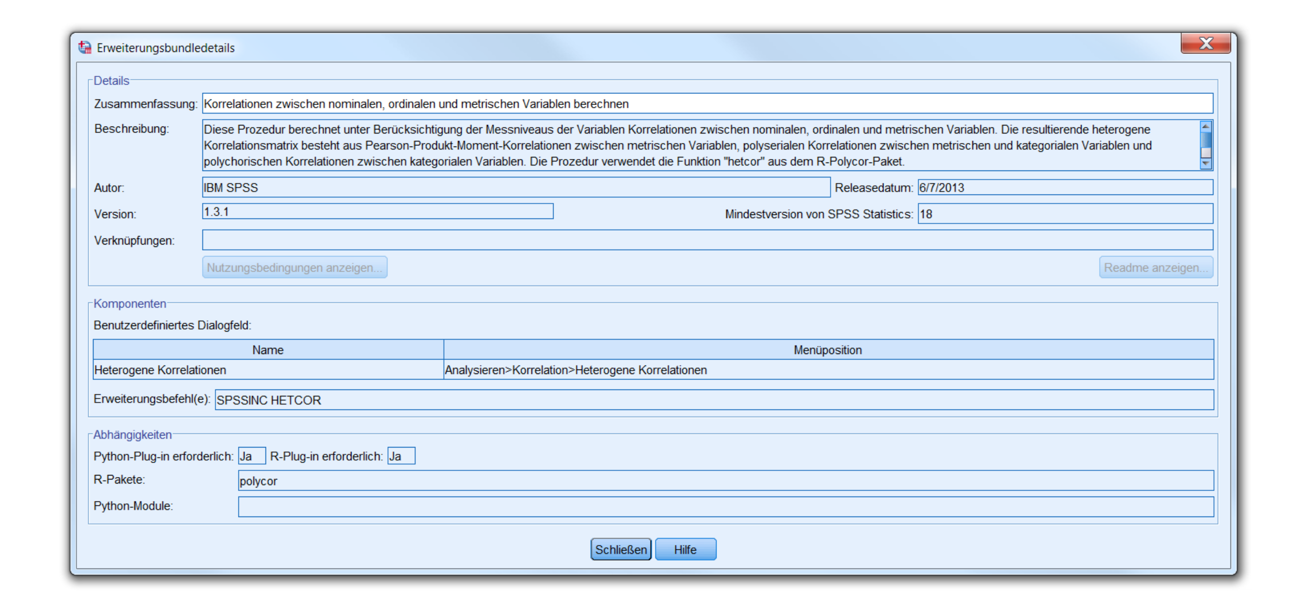
sowie der folgende Ordner mit den benutzerdefinierten Dialogen:



Um Informationen über die bereits installierten Erweiterungsbundles zu erhalten, wählt man den Menübefehl

Extras > Erweiterungsbundles > Installierte Erweiterungsbundles anzeigen

Im folgenden Beispiel ist für das (mit den R-Essentials installierte) Bundle SPSSINC HETCOR u.a. zu erfahren, dass sowohl das **Python**- als auch das **R-Plug-in** erforderlich ist:



Benötigt ein Bundle zusätzliche **R**-Pakete, versucht der Bundle-Installer, diese aus dem Internet herunter zu laden und zu installieren, was nur unter den folgenden Voraussetzungen gelingen kann:

- **Administratorrechte**
  Um die in der Regel (je nach dem **R**-Installationsordner) erforderlichen Administratorrechte bereit zu stellen, muss SPSS Statistics mit dem Administratorkonto ausgeführt werden. Bei einer Windows-Version ab Vista wählt man zum Starten von SPSS aus dem Kontextmenü zur Startverknüpfung den Eintrag **Als Administrator ausführen**.
- **Internetkontakt**
  Hat der Zielrechner keinen Internetkontakt, müssen die **R**-Pakete vor der Bundle-Installation mit Hilfe von Dateien über **R**-Verfahren installiert werden (vgl. Abschnitt 5.2.2).

Eine kurze Funktionsbeschreibung der von IBM SPSS erstellten Erweiterungsbundles bietet Cohen (2013).

2.3 Benutzerdefinierte Dialoge

2.3.1 Inhalt

Neben dem Erweiterungsbundle, das einen benutzerdefinerten Dialog und/oder ein Erweiterungskommando enthalten kann, existiert als kleineres Distributionsformat der „isolierte“ benutzerdefinierte Dialog. Er präsentiert eine bequeme Bedienoberfläche und realisiert seine Funktionalität entweder durch eine SPSS-interne Programmieroption (z.B. Makrotechnik, Programmiersprache MATRIX) oder durch eine externe Programmieroption (z.B. **R** oder Python). Es fehlt die Möglichkeit zur Integration in SPSS-Programme, doch die in vielen Situationen bzw. von vielen Benutzern bevorzugte interaktive Nutzung wird perfekt unterstützt.

2.3.2 Erstellung

Wer mit SPSS ein benutzerdefiniertes Dialogfeld erstellen möchte, steigt mit dem folgenden Menübefehl ein:

Extras > Benutzerdefinierte Dialogfelder > Dialogfelderstellung

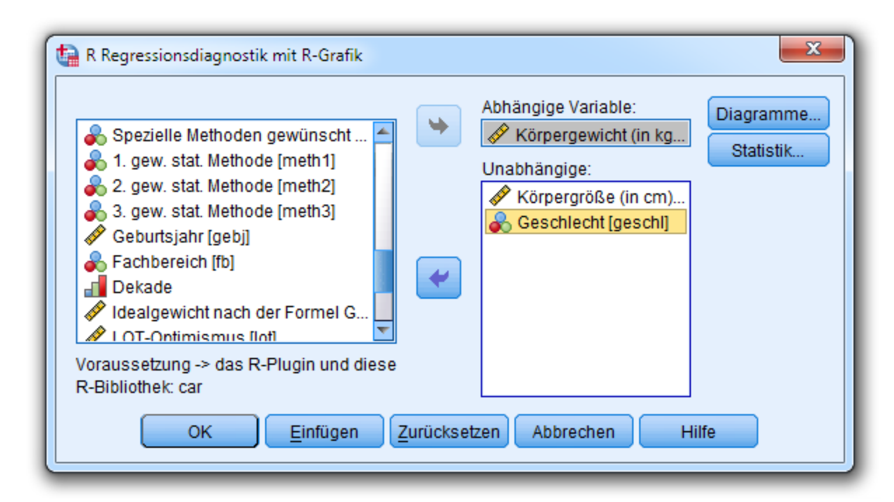
Es resultiert eine **Custom Dialog Package** - Datei mit der Namenserweiterung **SPD**. Eine Anleitung findet sich z.B. im Core-System Benutzerhandbuch (IBM SPSS 2013, Kapitel 20).

2.3.3 Installation

Viele freundliche Menschen haben benutzerdefinierte Dialoge erstellt und dabei oft **R** zur Realisation verwendet. Auf der folgenden Webseite:

http://gruener.userpage.fu-berlin.de/spss-dialogs.htm

bietet Hans Grüner vom Rechenzentrum der FU Berlin diverse benutzerdefinierte Dialoge mit **R**-Implementierung an, z.B. zur grafischen Diagnose von linearen Regressionsmodellen:



Wer einen benutzerdefinierten Dialog per SPD-Datei bezogen hat und nun installieren möchte, wählt folgenden Menübefehl:

Extras > Benutzerdefinierte Dialogfelder > Benutzerdefiniertes Dialogfeld installieren

Wird SPSS mit Administratorrechten ausgeführt, steht das benutzerdefinierte Dialogfeld anschließend allen Benutzern zur Verfügung. Per Voreinstellung erfolgt die Installation unter Windows 7 in einen Unterordner von

C:\Program Files\IBM\SPSS\Statistics\22\ext\lib

Wird SPSS mit normalen Benutzerrechten ausgeführt, steht das Dialogfeld anschließend nur dem Installateur zur Verfügung. Per Voreinstellung erfolgt die Installation unter Windows 7 beim Benutzer Otto in einen Unterordner von

C:\Users\Otto\AppData\Local\IBM\SPSS\Statistics\22\CustomDialogs.

Benötigt ein benutzerdefiniertes Dialogfeld zusätzliche **R**-Pakete, wird versucht, diese aus dem Internet herunter zu laden und zu installieren, wobei Administratorrechte unter eine Internetverbindung erforderlich sind (siehe Abschnitt 2.2).

3 Erweiterungsbundles in den R-Essentials

In diesem Abschnitt nutzen wir einige mit den **R**-Essentials installierte Erweiterungsbundles, die allesamt über ein Dialogfeld verfügen. Damit stehen relevante Funktionsergänzungen bei größtmöglicher Bequemlichkeit zur Verfügung, und wir können uns in diesem Abschnitt auf die Forschungsmethodik konzentrieren.

3.1 Robuste Regression

Techniken der robusten Regression kommen als Alternative zur OLS-Regression (***O****rdinary* ***L****east* ***S****quares*) in Frage, wenn problematische Einzelwerte die Interpretation erschweren:

- **Große Residuen**
  Hier geht es um Fälle mit extremen Residuen, die nicht auf Erhebungs- oder Erfassungsfehler zurückgehen und auch nicht überzeugend als außerhalb der Betrachtung liegend (z.B. zu anderen Populationen gehörig) entfernt werden können (Ausreißer).
- **Starke Hebel**
  Fälle mit extremen Werten bei unabhängigen Variablen verfügen über eine große Hebelwirkung und damit über einen oft unerwünscht starken Effekt auf die Schätz- und Testergebnisse.

Der Gesamteinfluss eines Falls auf die Ergebnisse ist im Wesentlichen ein Produkt aus der Hebelwirkung und der absoluten Größe des Residuums.

Wenn man die schädlichen Einflüsse von Ausreißern und Fällen mit großer Hebelwirkung auf die Schätz- und Testergebnisse ebenso vermeiden möchte wie den Verdacht, unliebsame Daten unterschlagen zu haben, können Techniken der robusten Regression ein Ausweg sein.

Im vorzustellenden Erweiterungsbundle sowie in der zugrunde liegenden **R**-Funktion **rlm()** aus dem **R**-Paket **MASS** kommen so genannte **M-Schätzer** zum Einsatz, die ...

- zwar den Einfluss von Fällen mit großen Residuen begrenzen,
- jedoch *nicht* robust sind gegenüber Fällen mit unauffälligen Residuen, aber große Hebelwirkung.

Die Verfahren mit M-Schätzer arbeiten als WLS-Regression (gewichtete Kleinst-Quadrat-Regression) mit iterativ verbesserten Gewichten (IDRE UCLA 2014a):

- Aufgrund der Residuen werden neue Gewichte berechnet (je größer der Residualbetrag, desto kleiner das Gewicht).
- Mit den neuen Gewichten erhält man aktualisierte Parameterschätzungen und neue Residuen.

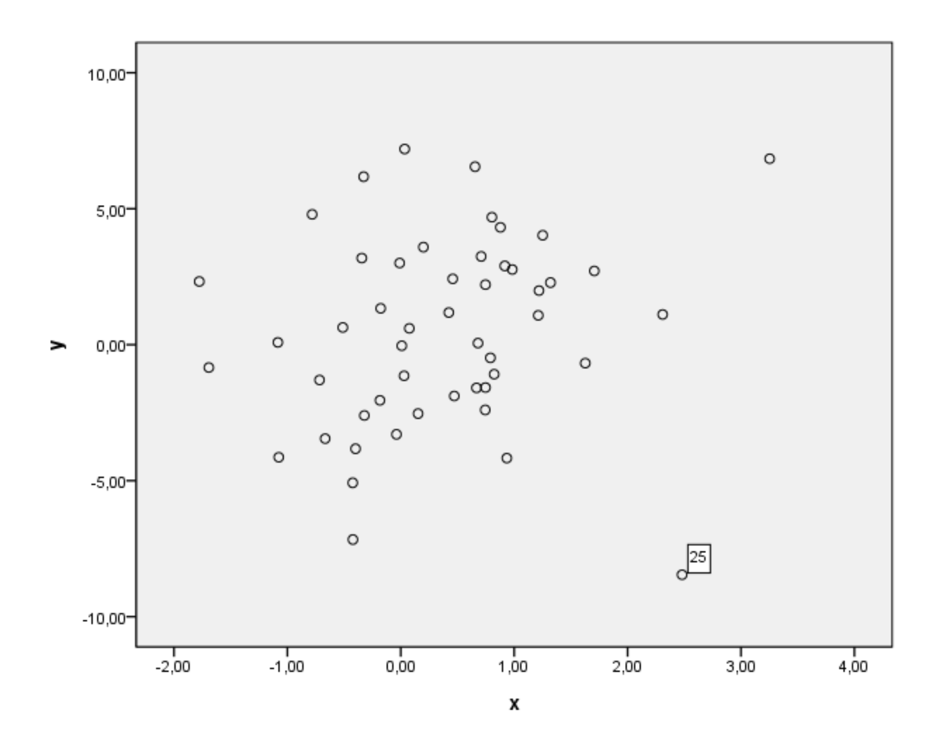
Wenn sich die Parameterschätzer nicht mehr ändern, stoppt der iterative Algorithmus.

Eine ausführliche Behandlung der robusten Regression bietet z.B. Ryan (1997).

Wir betrachten als Anwendungsbeispiel synthetische Daten mit einer bivariaten Regression und einem Ausreißer, der die Schätzung und Testung empfindlich stört. Das wahre Modell:

$$Y_i = 1 \cdot X_i + \varepsilon_i, \quad \text{mit } \operatorname{Var}(\varepsilon_i) = 9, \operatorname{Cov}(\varepsilon_i, \varepsilon_j) = 0$$

In der Stichprobe mit n = 50 befindet sich ein Fall (Nr. 25) mit dem „Störimpuls“ -13 im Y-Messwert, was im Streudiagramm gut zu erkennen ist:



In der OLS-Regression erhält man folgende Schätz- und Testergebnisse:

Modellzusammenfassung[b]

Modell	R	R-Quadrat	Korrigiertes R-Quadrat	Standardfehler des Schätzers
1	,172[a]	,030	,010	3,49338

a. Einflußvariablen : (Konstante), x
b. Abhängige Variable: y

Koeffizienten[a]

Modell		Nicht standardisierte Koeffizienten		Standardisierte Koeffizienten			Konfidenzintervall für B (95,0%)	
		B	Standardfehler	Beta	t	Sig.	Untergrenze	Obergrenze
1	(Konstante)	,245	,528		,465	,644	-,816	1,306
	x	,613	,505	,172	1,213	,231	-,403	1,628

a. Abhängige Variable: y

Die Schätzung zum Steigungskoeffizienten ist deutlich gemindert (0,61 statt 1,00), und der Signifikanztest erlaubt es nicht, die (falsche) Nullhypothese zu verwerfen (p = 0,231 > 0,05).

In der Tabelle mit den Cook-Distanzen zur Beurteilung der Einflussstärke von Einzelfällen[1] erreicht der Fall 25 den größten Wert (zur Definition der Cook-Distanzen siehe z.B. Baltes-Götz 2014a, Abschnitt 3.1.3):

[1] In SPSS per Syntax anzufordern mit dem folgenden Subkommando der Regressionsprozedur:

```
/RESIDUALS outliers(cook)
```

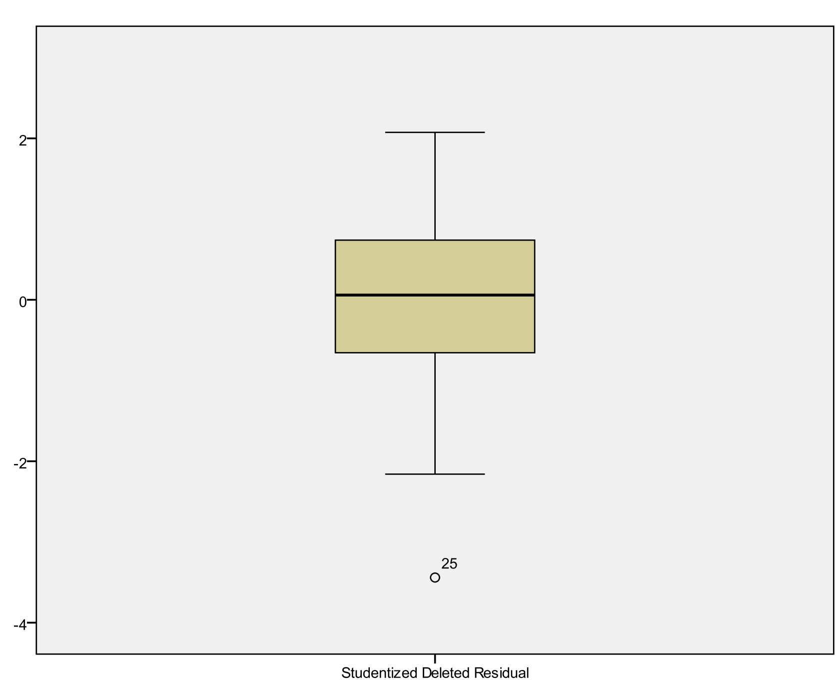
Ausreißerstatistik[a]

		Fallnummer	Statistik	Sig. F
Cook-Distanz	1	25	,617	,544
	2	2	,259	,773
	3	17	,074	,929
	4	8	,061	,941
	5	36	,054	,947
	6	29	,049	,952
	7	6	,046	,955
	8	39	,041	,960
	9	31	,037	,964
	10	24	,032	,968

a. Abhängige Variable: y

Als kritische Cook-Distanzen werden in der Literatur die Werte 1 (z.B. Weisberg 1985) und 4/*n* (z.B. Gordon 2010, S. 367) genannt, wobei wohl die zweite, strengere Grenze herangezogen werden sollte. In unserem Beispiel ergibt sich der Grenzwert 0,08, den der kritische Fall 25 deutlich überschreitet.

Das ausgelassen-studentisierte Residuum von -3,441 für Fall 25 deutet auf einen ernst zu nehmenden Ausreißer hin (zur Definition der ausgelassen-studentisierten Residuen siehe z.B. Baltes-Götz 2014a, Abschnitt 1.7.2):



Entfernt man den Fall 25 aus der Analyse, liefert die OLS-Regression ein deutlich verschiedenes Ergebnisbild:

Modellzusammenfassung

Modell	R	R-Quadrat	Korrigiertes R-Quadrat	Standardfehler des Schätzers
1	,323[a]	,104	,085	3,15529

a. Einflußvariablen : (Konstante), x

Koeffizienten[a]

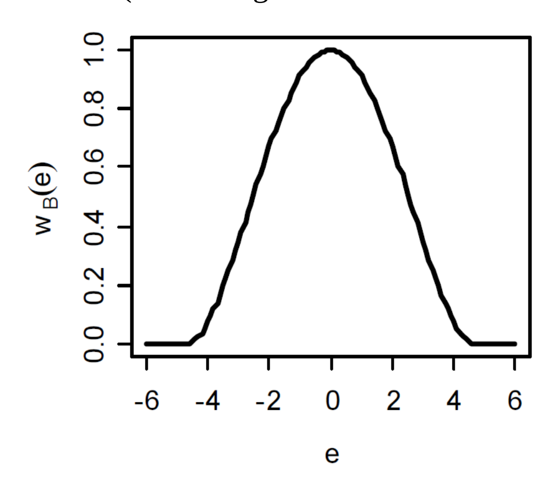
Modell		Nicht standardisierte Koeffizienten		Standardisierte Koeffizienten			Konfidenzintervall für B (95,0%)	
		B	Standardfehler	Beta	t	Sig.	Untergrenze	Obergrenze
1	(Konstante)	,289	,477		,606	,548	-,670	1,248
	x	1,122	,480	,323	2,339	,024	,157	2,087

a. Abhängige Variable: y

Die SPSS-Erweiterung zur robusten Regression ist verfügbar über den Menübefehl[1]

Analysieren > Regression > Robuste Regression

und verwendet die Funktion **rlm()** aus dem **R**-Paket **MASS**. Dabei wählt SPSS von den verfügbaren robusten Schätzmethoden den so genannten *biquadratischen Schätzer* (engl.: *bisquare estimator*), der folgende Gewichtungsfunktion verwendet (Abbildung übernommen aus Fox 2002, S. 3):



Der Einfluss eines Falles wird mit wachsendem Betrag seines Residuums reduziert, wobei auch kleine Abweichungsbeträge bereits zu einer schwachen Minderung führen.

Für die Beispieldaten (*inkl*. Fall 25) resultiert die folgende Tabelle mit **rlm()** - Ergebnissen:

Koeffizienten

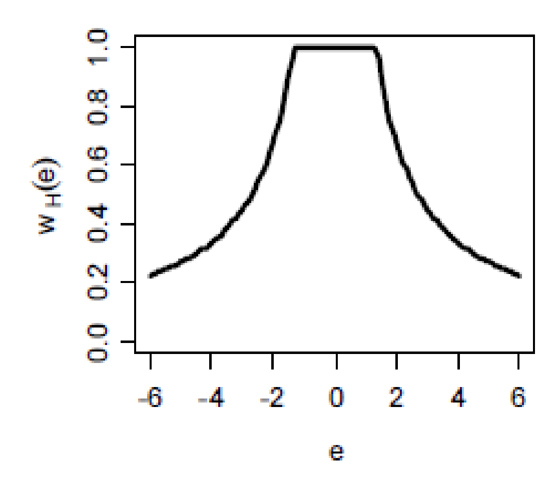
	Wert	Standard Fehler	t-Wert
(Konstanter Term)	,235	,524	,449
x	,955	,502	1,903

rlm(formula = y ~ x, data = dta, na.action = na.exclude, method = "MM", model = FALSE)
Residuum-Standardfehler: 3.48643
Freiheitsgrade: 48

Der Steigungskoeffizient wird sehr präzise geschätzt (wahrer Wert: 1). Leider werden zu den Regressionskoeffizienten mangels Vertrauen in die Verteilung der Prüfstatistik (berechnet als Quotient aus dem Schätzer und seinem Standardfehler) keine Überschreitungswahrscheinlichkeiten (*p*-Werte) geliefert. Ebenso fehlen Vertrauensintervalle. In Abschnitt 4.2 wird demonstriert, wie man mit Hilfe von **R**-Syntax approximative *p*-Werte ermitteln kann.

Ist an Stelle des biquadratischen Schätzers, den SPSS in der Erweiterungsprozedur verwendet, Hubers M-Schätzer mit der folgenden Gewichtungsfunktion (Abbildung übernommen aus Fox 2002, S. 3)

[1] Bis zur Version 1.2.1 des Erweiterungsbundles zur robusten Regression hieß das Menüitem ***Solide Regression***.



gewünscht, kann man in einem **R**-Syntax - Block die Variablen der SPSS-Arbeitsdatei an **R** übergeben und die **R**-Funktion **rlm()** direkt verwenden (siehe Abschnitt 4.2). Bei der Huber-Gewichtung muss der Residuumsbetrag eine Grenze überschreiten, bevor die Minderung des Einflussgewichts einsetzt. Im Unterschied zum biquadratischen Schätzer bleibt der Einfluss auch bei Fällen mit sehr großem Residuumsbetrag größer 0.

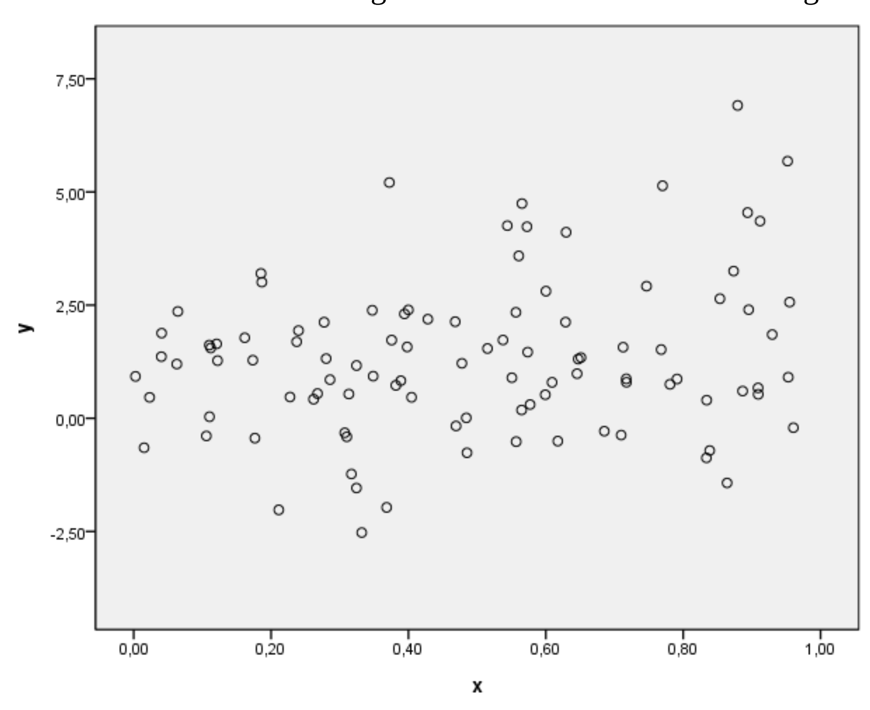
3.2 Breusch-Pagan - Heteroskedastizitäts-Test

Der Breusch–Pagan - Test prüft für lineare Regressionsmodelle die Nullhypothese homogener Fehlervarianzen.

Wir betrachten als Anwendungsbeispiel synthetische Daten mit einer bivariaten Regression und Fehlervarianzen mit ausgeprägter Abhängigkeit vom Wert des Regressors:

$$Y_i = 2 \cdot X_i + \varepsilon_i, \quad \text{mit } \mathrm{Var}(\varepsilon_i) = (1 + X_i)^2, \mathrm{Cov}(\varepsilon_i, \varepsilon_j) = 0, X_i \text{ gleichverteilt auf } [0, 1]$$

In der Stichprobe mit n = 100 ist die mit dem Regressor wachsende Fehlervarianz gut zu erkennen:



Das Erweiterungskommando zur robusten Regression ist verfügbar über den Menübefehl

Analysieren > Regression > Residuums-Heteroskedastizitätstest

und verwendet die Funktion **ncvTest** aus dem **R**-Paket **car**.

Im Beispiel wird die Heteroskedastizität erkannt:

Nichtkonstanter Varianzwerttest

	Chi-Quadrat	D.f	Sig.
Testergebnis	6,352	1,000	,012

Varianzmodell: fitted.values
Berechnet durch R-ncvTest-Funktion

Nach einem signifikanten Heterogenitätstest muss man übrigens den Versuch einer linearen Modellierung nicht aufgeben. Mittlerweile sind Verfahren zur robusten Inferenzstatistik trotz Heteroskedastizität entwickelt worden, die in **R** wie in SPSS zur Verfügung stehen (siehe z.B. Baltes-Götz 2014a, Abschnitt 1.7.3.4).

3.3 Polyseriale und polychorische Korrelationen

In der statistisch-empirischen Forschung sind häufig *ordinale* Variablen im Sinne von vergröbernden Messungen von latenten Variablen mit metrischer Skalenqualität und approximativer Normalverteilung anzutreffen. Oft ist man an den Korrelationen zwischen den latenten Merkmalen interessiert, hat aber nur die ordinalen Indikatoren zur Verfügung. Mit der Pearson-Formel zur Korrelationsberechnung erhält man aus den ordinalen Maßen mehr oder weniger stark verzerrte Schätzer (siehe unten).

Stimmt für zwei ordinale Indikatoren die Annahme, dass sie durch vergröbernde Messungen aus einer bivariaten Normalverteilung der eigentlich interessierenden latenten Variablen hervorgegangen sind, dann kommt das **polychorische** Schätzverfahren der gesuchten Korrelation näher als die Pearson-Formel.

Ist von den beiden zu korrelierenden Merkmalen nur eines vergröbernd gemessen worden, sodass ein intervallskalierter Indikator auf einen ordinalen trifft, erbringt die **polyseriale** Schätzformel eine analoge Reparaturleistung.

Als Anwendungsbeispiel betrachten wir latente Variablen *X* und *Y* mit einer wahren Korrelation von 0,5:[1]

$$Y_i = 0{,}5 \cdot X_i + \varepsilon_i \quad \text{mit } \varepsilon_i \sim \mathrm{N}(0; 0{,}75) \text{ und } \mathrm{Cov}(\varepsilon_i, \varepsilon_j) = 0;\ X_i \sim \mathrm{N}(0;\ 1)$$

Daraus entstehen durch vergröberndes Messen die ordinalen Indikatoren `X3` und `Y3` mit jeweils 3 Ausprägungen, z.B. mit Hilfe der folgenden SPSS-Syntax:

```
RECODE X (LO THRU -1 = 1) (-1 THRU 1 = 2) (ELSE = 3) INTO X3.
RECODE Y (LO THRU 0 = 1) (0 THRU 2 = 2) (ELSE = 3) INTO Y3.
VARIABLE LEVEL  X3, Y3 (ORDINAL).
```

Verwendet man die SPSS-Prozedur zur Berechnung von Pearson-Korrelationen, dann resultieren deutlich geminderte Schätzungen für die interessierende Korrelation von *X* und *Y* durch die Korrelationen von `X3` und `Y3` (beide Variablen ordinal) bzw. `X` und `Y3` (nur `Y3` ordinal):

[1] $$\mathrm{Cor}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\,\mathrm{Var}(Y)}} = \frac{\mathrm{Cov}(X, 0{,}5X + \varepsilon)}{\sqrt{\mathrm{Var}(0{,}5X + \varepsilon)}} = \frac{0{,}5}{\sqrt{0{,}25 + \mathrm{Var}(\varepsilon)}} = \frac{0{,}5}{\sqrt{0{,}25 + 0{,}75}} = 0{,}5$$

Korrelationen

		X	Y	X3	Y3
X	Korrelation nach Pearson	1	,526	,848	,389
	Signifikanz (2-seitig)		,000	,000	,000
	N	500	500	500	500
Y	Korrelation nach Pearson	,526	1	,446	,827
	Signifikanz (2-seitig)	,000		,000	,000
	N	500	500	500	500
X3	Korrelation nach Pearson	,848	,446	1	,343
	Signifikanz (2-seitig)	,000	,000		,000
	N	500	500	500	500
Y3	Korrelation nach Pearson	,389	,827	,343	1
	Signifikanz (2-seitig)	,000	,000	,000	
	N	500	500	500	500

Das Erweiterungskommando zur Berechnung von polychorischen und polyserialen Korrelationen mit Hilfe von **R** (und Python) ist verfügbar über den Menübefehl

Analysieren > Korrelation

Im Beispiel erhalten wir im polychorischen und im polyserialen Fall deutlich verbesserte Schätzungen in der Nähe des wahren Wertes:

Pearson-, polyseriale und polychorische Korrelationen

		Variablen	
Variablen	Statistik	X3	Y3
X3	Korrelation	1,000	,494
	Standard Fehler	,000	,052
	X	500,000	500,000
Y3	Korrelation	,494	1,000
	Standard Fehler	,052	,000
	X	500,000	500,000

Durch das R-Hetcor-Paket berechnete Korrelationen

Korrelationstypen

	Variablen	
Variablen	X3	Y3
X3		Polychoric
Y3	Polychoric	

Pearson-, polyseriale und polychorische Korrelationen

		Variablen	
Variablen	Statistik	X	Y3
X	Korrelation	1,000	,469
	Standard Fehler	,000	,043
	X	500,000	500,000
Y3	Korrelation	,469	1,000
	Standard Fehler	,043	,000
	X	500,000	500,000

Durch das R-Hetcor-Paket berechnete Korrelationen

Korrelationstypen

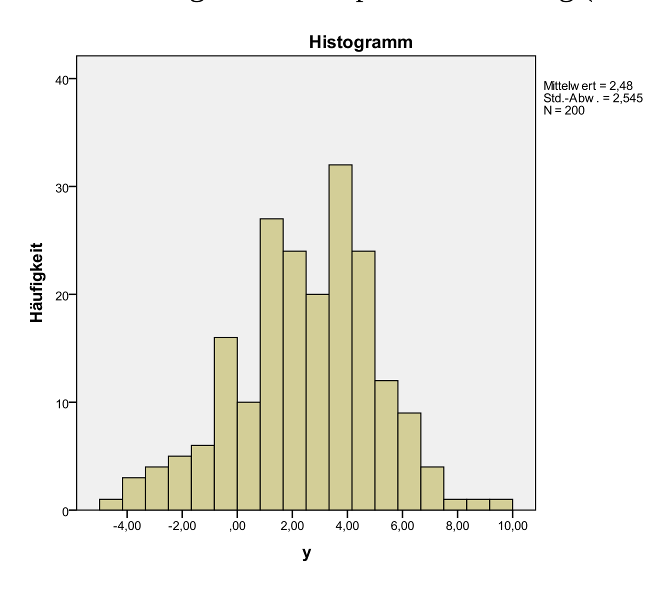
	Variablen	
Variablen	X	Y3
X		Polyserial
Y3	Polyserial	

3.4 Tobit-Regression

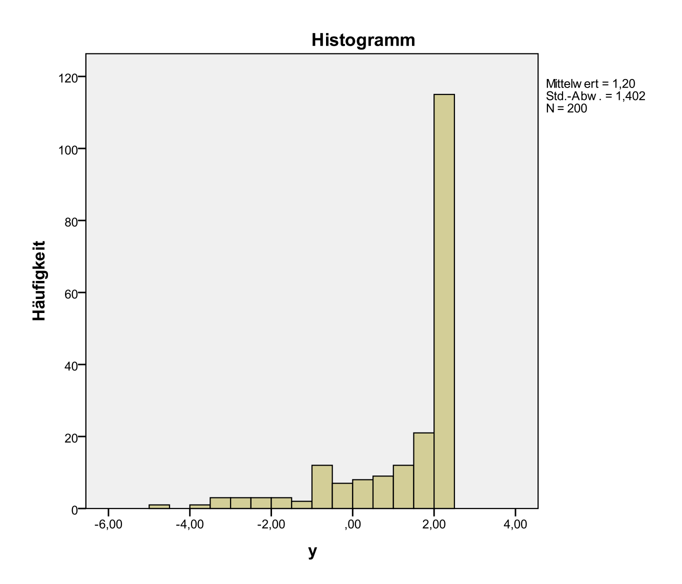
Die Tobit-Regression kommt als Alternative zur OLS-Regression bei zensierten Daten (siehe z.B. IRDE UCLA 2014b) in Frage. Als Anwendungsbeispiel betrachten wir eine bivariate Regression mit dem folgenden wahren Modell (mit Regressionsgewicht 1):

$$Y_i = 1 \cdot X_i + \varepsilon_i \quad \text{mit } \varepsilon_i \sim \text{N}(0;4) \text{ und } \text{Cov}(\varepsilon_i, \varepsilon_j) = 0$$

Aus der Kriteriumsvariablen mit der folgenden Stichprobenverteilung (n = 200)



entsteht eine rechts-zensierte Variante, indem alle Werte größer oder gleich 2 auf den Randwert 2 gesetzt werden:



Bei einer OLS-Regression (*Ordinary Least Squares*) mit dem zensierten Kriterium ist die Schätzung des Regressionskoeffizienten erheblich verzerrt:

Koeffizienten[a]

Modell		Nicht standardisierte Koeffizienten		Standardisierte Koeffizienten	t	Sig.	Konfidenzintervall für B (95,0%)	
		B	Standardfehler	Beta			Untergrenze	Obergrenze
1	(Konstante)	,098	,181		,539	,590	-,260	,456
	x	,438	,063	,443	6,959	,000	,314	,562

a. Abhängige Variable: y

Das Erweiterungskommando zur Berechnung einer Tobit-Regression mit Hilfe von **R**-Funktionen ist verfügbar über den Menübefehl[1]

Analysieren > Regression > Tobit-Regression

Es resultiert eine Schätzung nahe beim wahren Wert 1:

Koeffizienten

	Koeffizient	Standard Fehler	z-Wert	Sig.
(Konstanter Term)	-,101	,353	-,285	,776
x	1,068	,151	7,092	,000
Log(-Skalierung)	,806	,084	9,614	,000

Untergrenze: Keines, Obergrenze: 2
tobit(formula = y ~ x, left = -Inf, right = 2, dist = "gaussian", data = dta, na.action = na.exclude)
Saklierung: 2.2380
Residuum d.f.: 197
Log-Likelihood: -274.573 D.f.: 3
Wald-Statistik: 50.290 D.f.: 1

Neben dem Regressionsgewicht wird unter der Bezeichnung **Skalierung** noch ein Koeffizient mitgeteilt, welcher die Standardabweichung der Residuen schätzt und im konkreten Fall nahe bei dem Wert liegt, den eine OLS-Regression für die *unzensierten* Daten ermittelt (Bezeichnung **Standardfehler des Schätzers** in der Tabelle **Modellzusammenfassung**):

Modellübersicht[b]

Modell	R	R-Quadrat	Angepasstes R-Quadrat	Standardfehler der Schätzung
1	,586[a]	,344	,341	2,06634

a. Prädiktoren: (Konstante), x

b. Abhängige Variable: y

In die deutsche Übersetzung der Tobit-Ausgabe haben sich zwei Fehler eingeschlichen:

- Die Tabellenzeile mit der logarithmierten Skalierung enthält einen überflüssigen Bindestrich, der bei Deutung als Minuszeichen zu einem groben Fehler führt. Der angezeigte Wert ist korrekt:

  $$\text{Log}(2{,}238) = 0{,}806$$

- In der Fußnote muss es *Skalierung* heißen statt *Saklierung*.

Die englische Tobit-Ausgabe ist fehlerfrei:

[1] In **R** sind verschiedene Tobit-Lösungen vorhanden (z.B. in den Paketen **AER**, **VGAM** und **censReg**). SPSS verwendet die Funktion **tobit()** aus dem Paket **AER**.

Coefficients

	Coefficient	Std. Error	z Value	Sig.
(Intercept)	-,101	,353	-,285	,776
x	1,068	,151	7,092	,000
Log(scale)	,806	,084	9,614	,000

Lower bound: None, Upper bound: 2
tobit(formula = y ~ x, left = -Inf, right = 2, dist = "gaussian", data = dta, na.action = na.exclude)
Scale: 2.2380
Residual d.f.: 197
Log likelihood: -274.573 D.f.: 3
Wald statistic: 50.290 D.f.: 1

4 R-Funktionen über das SPSS-Syntaxfenster nutzen

Um **R**-Funktionen zu nutzen, die *nicht* über SPSS-Erweiterungskommandos oder benutzerdefinierte Dialoge erschlossen sind, ist **R**-Syntax zu verfassen. Die Übergabe von SPSS-Variablen an **R** und der Rücktransport von **R**-Ergebnissen fällt aber nicht schwer. Man erstellt in einem SPSS-Syntaxfenster einen Block mit **R**-Syntax, eingerahmt durch die SPSS-Kommandos `BEGIN PROGRAM R` und `END PROGRAM`. In der **R**-Syntax spielen Funktionen aus dem **R**-Paket zu SPSS, das mit den **R**-Essentials automatisch installiert wird, eine wesentliche Rolle, z.B. Funktion **spssdata.GetDataFromSPSS()** für den Zugriff auf die Variablen der SPSS-Arbeitsdatei. Eine Dokumentation dieser **R**-Funktionen zur SPSS-Unterstützung finden Sie im Dokument

R Integration Package for IBM SPSS Statistics

Es ist nach der Installation der **R**-Essentials über das Hilfemenü in SPSS Statistics verfügbar:

Hilfe > Programmierbarkeit > R-Plugin

Das **R**-Paket zur SPSS-Unterstützung wird bei der **R**-Nutzung via SPSS automatisch geladen, so dass ein explizites Laden mit der **R**-Funktion **library()** (vgl. Abschnitt 5.2.1) *nicht* erforderlich ist.

Bei der in diesem Abschnitt vorgestellten Arbeitsweise sind nur wenige Kenntnisse der **R**-Syntax erforderlich, so dass wir eine Behandlung von **R** als Programmiersprache bis zum Abschnitt 5 aufschieben.

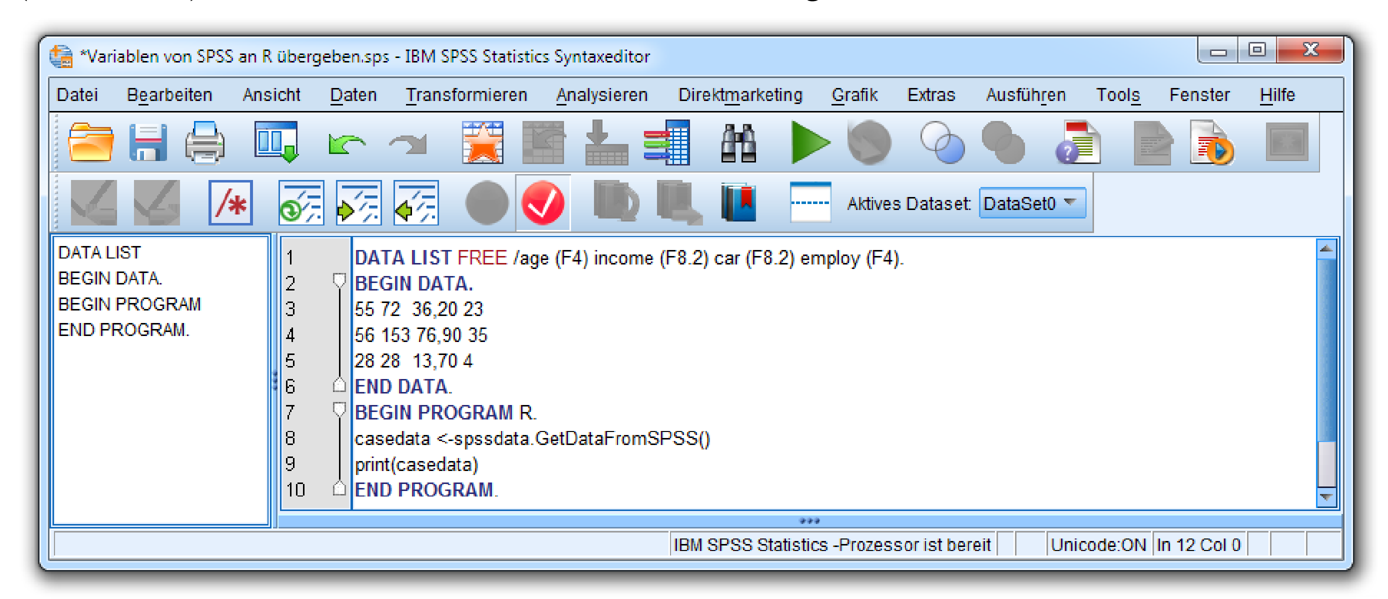
Sollte eine **R**-Anweisung nicht in eine Zeile passen, kann sie auf Folgezeilen fortgesetzt werden, z.B.

```
BEGIN PROGRAM R.
dta <- spssdata.GetDataFromSPSS(factorMode="labels")
boxplot(dta$aergo ~ dta$geschl, col="lightblue3", varwidth=TRUE, boxwex=0.75,
        xlab="Geschlecht", ylab="Ärger ohne KFA")
END PROGRAM.
```

4.1 SPSS-Variablen an R übergeben

4.1.1 Übergabe der kompletten Arbeitsdatei

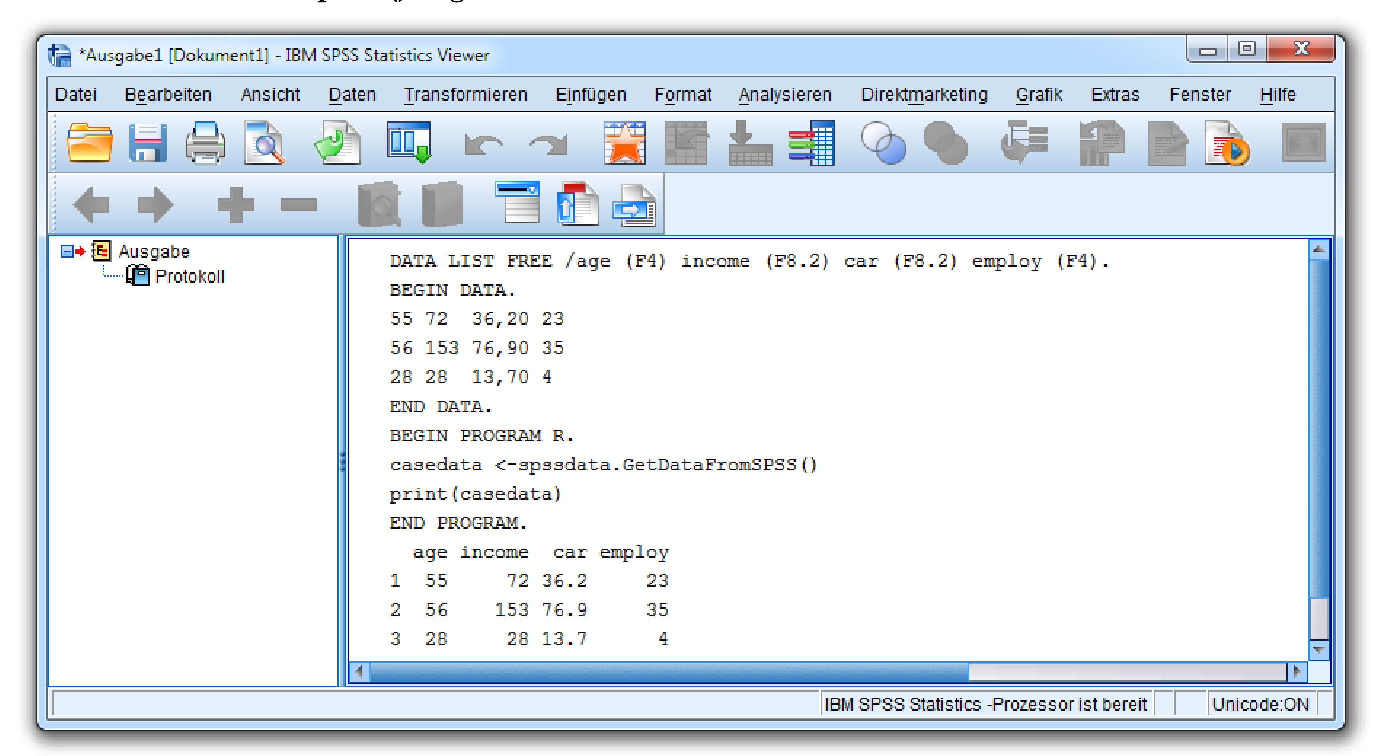
Mit Hilfe der Funktion **spssdata.GetDataFromSPSS()**, die im **R**-Paket **spss220** enthalten ist, kann man in **R** leicht auf die Variablen der SPSS-Arbeitsdatei zugreifen. Im folgenden Beispiel aus Levesque (2011, S. 341) werden *alle* Variablen der Arbeitsdatei übertragen:



In **R** resultiert ein so genannter **Data Frame** (deutsch: *Datentabelle*, vgl. Abschnitt 5.3.4.7), der einem SPSS-Datenblatt weitgehend entspricht.

Der **Data Frame** wird mit dem Operator "<-" der **R**-Variablen (man sagt auch: *dem **R**-Objekt*) `casedata` zugewiesen.

Über die **R**-Funktion **print()** kann man die in einer Datentabelle vorhandenen Variablen ausgeben lassen.[1] Im Beispiel erhält man im SPSS-Ausgabefenster eine Protokollausgabe der SPSS- und **R**-Kommandos sowie das **print()**-Ergebnis:



Deutsche Umlaute in Variablennamen oder Wertbeschriftungen machen bei der Datenübergabe an **R** *keinen* Ärger.

4.1.2 Eine Auswahl von SPSS-Variablen übergeben

Um eine Auswahl der SPSS-Variablen in der aktuellen Arbeitsdatei an **R** zu übergeben, verwendet man in der **R**-Funktion **spssdata.GetDataFromSPSS()** das Argument **variables**, dem ein Vektor mit Variablennamen zu übergeben ist. Diesen Vektor bildet man in der Regel über die **R**-Funktion **c()**. Weil diese Funktion sehr oft benötigt wird, hat sie einen kurzen Namen erhalten, wobei das *c* für *combine* oder *concatenate* steht. Die folgende Syntax:

```
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(variables=c("age","income", "car"))
print(casedata)
END PROGRAM.
```

liefert aufgrund der eben vorgestellten SPSS-Arbeitsdatei die Tabelle:

[1] Die Funktion **print()** wird hier der Deutlichkeit halber explizit notiert. Der Variablenname `casedata` genügt eigentlich, weil er von **R** als impliziter Aufruf der Funktion **print()** verstanden wird.

```
  age income  car
1  55     72 36.2
2  56    153 76.9
3  28     28 13.7
```

In **R** ist die **Groß-/Kleinschreibung** signifikant und muss auch bei SPSS-Variablennamen beachtet werden. Daher würde das folgende Kommando zu einem Fehler führen:

```
casedata <- spssdata.GetDataFromSPSS(variables=c("Age","income", "car"))
```

Hingegen besteht einige syntaktische Freiheit beim Verfassen der Variablenliste, so dass folgende Varianten erlaubt sind:

a) Eine gemeinsame Zeichenfolge mit Leerzeichen zwischen den Variablennamen:

```
casedata <- spssdata.GetDataFromSPSS(variables=c("age income car"))
```

b) Mit dem Schlüsselwort **TO** bildet man eine Liste von Variablen, die in der Arbeitsdatei (dem aktiven Datenblatt) hintereinander stehen:

```
casedata <- spssdata.GetDataFromSPSS(variables=c("age to car"))
```

Beim Schlüsselwort **TO** ist die Groß-/Kleinschreibung beliebig. Selbstverständlich dürfen mit TO gebildete Sequenzen zusammen mit Einzelvariablen aufgelistet werden.

c) Anstelle der Namen kann man die Positionen der Variablen im SPSS-Datenblatt angeben:

```
casedata <- spssdata.GetDataFromSPSS(variables=c(0, 1, 2))
```

Es ist zu beachten, dass die Elemente mit 0 beginnend nummeriert werden.

d) Bei einer Liste von aufeinanderfolgenden Positionen ist der **R**-Sequenzoperator erlaubt (siehe Abschnitt 5.3.4.2.1):

```
casedata <- spssdata.GetDataFromSPSS(variables=0:2)
```

4.1.3 Variablen in einer R - Datentabelle ansprechen

Auf eine einzelne Variable in einer **R** - Datentabelle kann man über die per $-Zeichen verbundene Kombination aus dem Datentabellen- und dem Variablennamen zugreifen, z.B.:

```
BEGIN PROGRAM R.
casedata$age
END PROGRAM.
```

4.1.4 Persistenz und Löschen von R-Objekten

Während SPSS große Datensätze nur teilweise im Arbeitsspeicher (RAM) hält, befindet sich ein **R** - Data Frame stets komplett im Speicher.

Das Beispiel in Abschnitt 4.1.3 zeigt außerdem, dass der **R**-Workspace (mit allen Variablen) innerhalb einer SPSS-Sitzung zwischen zwei **R**-Einsätzen erhalten bleibt, d.h.:

- Die in früheren **R**-Blöcken angelegten Objekte stehen weiter zur Verfügung.
- Eventuell ist es sinnvoll, belegten Speicherplatz (im RAM) durch explizites Entfernen von Objekten mit der **R**-Funktion **rm()** (alias **remove()**) wieder frei zu geben, z.B.:

```
BEGIN PROGRAM R.
  .  .  .
rm(casedata)
  .  .  .
END PROGRAM.
```

Weil SPSS nicht darauf angewiesen ist, alle Daten im Hauptspeicher (RAM) zu halten, müssen SPSS-Anwender nicht über das Einsparen von Speicherplatz nachdenken. **R** hingegen muss die zu analysierenden Daten komplett im Hauptspeicher halten, so dass es empfehlenswert ist, überflüssige Objekte explizit zu entfernen (Muenchen 2011, S. 33).

4.1.5 Kategoriale SPSS-Variablen als Faktoren an R übergeben

Ein (ordinaler) Faktor in **R** (siehe Abschnitt 5.3.4.3) ist das Analogon zu einer kategorialen (nominalen oder ordinalen) SPSS-Variablen.

SPSS-Variablen mit dem Datentyp Zeichenfolge (String) werden in der **R** - Datentabelle als Faktoren abgelegt. Aus numerischen SPSS-Variablen resultieren per Voreinstellung numerische Vektoren. Das gilt auch für numerische SPSS-Variablen mit dem Messniveau Nominal oder Ordinal.

Durch Verwendung des Arguments **factorMode** im Aufruf der Funktion **spssdata.GetDataFromSPSS()** veranlasst man, dass

- nominale numerische SPSS-Variablen in **R** zu Faktoren und
- ordinale numerische SPSS-Variablen in **R** zu *geordneten* Faktoren werden.

Mit dem zum Argument **factorMode** anzugebenden Wert legt man fest, ob bei einer kategorialen numerischen SPSS-Variablen die Werte oder die Werteetiketten in **R** zur Beschriftung der Faktorstufen verwendet werden sollen:

factorMode-Parameter	In **R** werden die Faktorstufen beschriftet über …
labels	die Werteetiketten der SPSS-Variablen
levels	die Werte der SPSS-Variablen

Aus der folgenden Syntax

```
DATA LIST FREE /y (F2) gruppe (F2) stufe (F2).
BEGIN DATA.
8  1  1
7  1  1
3  2  2
1  2  3
END DATA.
VARIABLE LEVEL y (SCALE) /gruppe (NOMINAL) /stufe(ORDINAL).
VALUE LABELS gruppe 1 'Jung' 2 'Alt' /stufe 1 'A' 2 'B' 3 'C'.

BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(factorMode="labels")
class(casedata$gruppe)
class(casedata$stufe)
str(casedata$gruppe)
END PROGRAM.
```

resultiert die Ausgabe:

```
[1] "factor"
[1] "ordered" "factor"
 Factor w/ 2 levels "Jung","Alt": 1 1 2 2
```

Per **class()** - Funktion erhält man die Auskunft, dass `casedata$gruppe` ein Faktor und `casedata$stufe` ein geordneter Faktor ist. Der **str()** - Ausgabe ist u.a. zu entnehmen, dass die Kategorien des **R**-Faktors `casedata$gruppe` gemäß **factorMode**-Spezifikation durch die SPSS-Wertelabels beschriftet sind.

4.1.6 Indikatoren für fehlende Werte

Bei numerischen SPSS-Variablen werden per Voreinstellung benutzerdefinierte Missing Data - Indikatoren (ab jetzt kurz: MD-Indikatoren) ebenso wie SYSMIS in der **R** – Datentabelle zum Wert **NaN** (*Not a Number*). Bei alphanumerischen SPSS-Variablen resultiert in **R** der Wert **NA** (*Not Available*). Das Ersetzen der benutzerdefinierten MD-Indikatoren ist erforderlich, weil es in **R** *nicht* möglich ist, beliebige Werte als Indikatoren für fehlende Werte zu deklarieren.

Das folgende Beispielprogramm

```
DATA LIST LIST (',') /numvar (F2) strvar (A1).
BEGIN DATA.
9,A
7,M
,B
END DATA.
MISSING VALUES numvar (9)  /strvar ('M').

BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS()
print(casedata)
END PROGRAM.
```

liefert die Ausgabe:

```
  numvar strvar
1    NaN      A
2      7   <NA>
3    NaN      B
```

Setzt man, wie es z.B. von Muenchen (2011, S. 34) empfohlen wird, in der Funktion **spssdata.GetDataFromSPSS()** das optionale Argument **missingValueToNA** auf den Wert **TRUE**,

```
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(missingValueToNA=TRUE)
print(casedata)
END PROGRAM.
```

dann resultiert bei fehlenden Werten von numerischen SPSS-Variablen in der **R** - Datentabelle der Wert **NA** (*Not Available*):

```
  numvar strvar
1     NA      A
2      7   <NA>
3     NA      B
```

Der MD-Indikator **NA** für die Variable `strvar` wird durch eckige Klammern begrenzt, weil die Variable alphanumerische Kategorien-Labels besitzt, und keine Textbegrenzungszeichen für die Ausgabe angefordert wurden (z.B. mit `print(casedata, quote = TRUE)`).

In der Regel werden in **R** bei numerischen Variablen die beiden Werte **NA** und **NaN** gleich behandelt, und die Funktion **is.na()** liefert für beide Werte ein TRUE. Wer gezielt auf den Wert **NaN** prüfen will, hat die **R**-Funktion **is.nan()** zur Verfügung, die bei **NaN** ein TRUE und bei **NA** ein **FALSE** liefert.

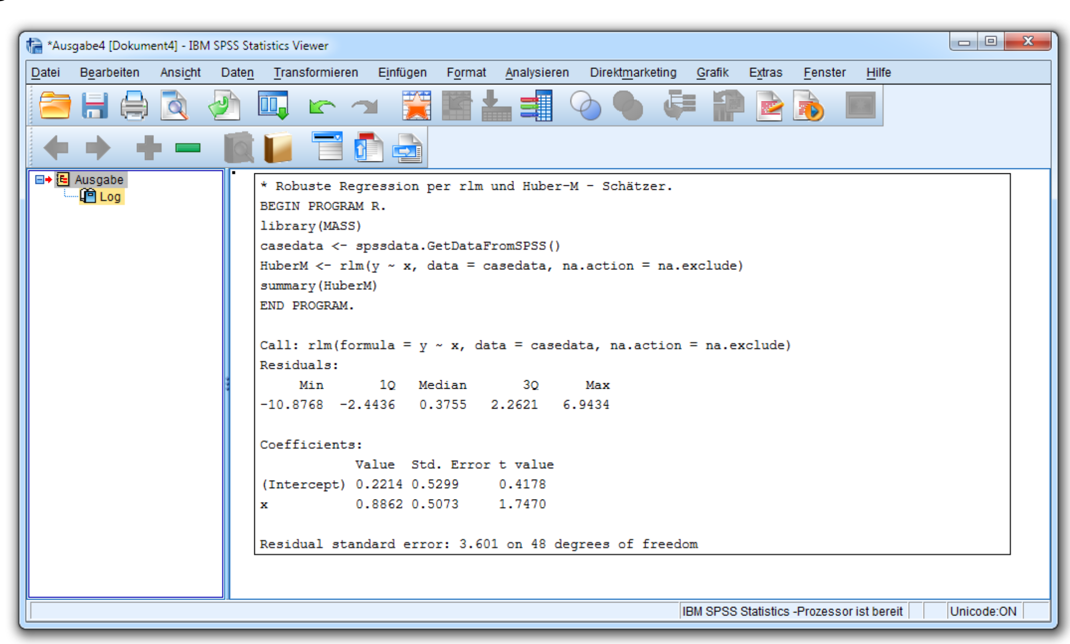
4.2 R-Auswertungsfunktionen verwenden und Ausgaben im SPSS-Viewer anzeigen

In Abschnitt 3.1 haben wir per SPSS-Erweiterungskommando eine robuste Regressionsanalyse mit der **R**-Funktion **rlm()** aus dem Paket **MASS** durchgeführt und dabei den biquadratischen M-Schätzer akzeptiert, den SPSS in der Erweiterungsprozedur verwendet. Wer stattdessen Hubers M-Schätzer bevorzugt, kann in einem **R**-Syntax - Block die Variablen der SPSS-Arbeitsdatei an **R** übergeben und die **R**-Funktion **rlm()** direkt verwenden, die per Voreinstellung mit dem M-Schätzer arbeitet:

```
BEGIN PROGRAM R.
library(MASS)
casedata <- spssdata.GetDataFromSPSS()
huber <- rlm(y ~ x, data = casedata)
summary(huber)
END PROGRAM.
```

Benötigte **R**-Funktionen befinden sich oft in Paketen, die zunächst über einen Aufruf der Funktion **library()** geladen werden müssen (siehe Beispiel).

Wie schon mehrfach zu sehen war, erscheint die Standardausgabe von **R** in einen **Log**-Abschnitt des SPSS-Ausgabefensters:



Im Programm wird die **rlm()** - Ausgabe in die Variable `huber` geleitet und von dort per **summary()** - Funktion in einiger Ausführlichkeit hervorgelockt. Auf direktem Weg

```
rlm(y ~ x, data = casedata)
```

erhält man nur eine spartanische Ausgabe:

```
Call:
rlm(formula = y ~ x, data = casedata, na.action = na.exclude)
Converged in 4 iterations

Coefficients:
(Intercept)           x
  0.2214148   0.8861699

Degrees of freedom: 50 total; 48 residual
Scale estimate: 3.6
```

Die zurückhaltende Originalausgabe von statistischen Auswertungsfunktionen und die Verwendung der Extraktionsfunktion **summary()** sind typisch für **R** (Muenchen 2011, S. 99f).

Die Koeffiziententabelle der **rlm()** - Ausgabe enthält (wie die in Abschnitt 3.1 präsentierte Pivot-Tabelle zum korrespondierenden Erweiterungskommando) *keine* Überschreitungswahrscheinlichkeiten (*p*-Levels) zu den Signifikanztests. Bei genügend Vertrauen in die approximative Normalverteilung der Prüfstatistiken kann man mit folgender **R**-Syntax die Überschreitungswahrscheinlichkeiten für einen ein- bzw. zweiseitigen Test gegen den theoretischen Wert 0 bestimmen (siehe Bellio & Ventura 2005, S. 16):

```
BEGIN PROGRAM R.
pst <- 1.747
psingle <- min(1-pnorm(pst), pnorm(pst))
psingle
2 * psingle
END PROGRAM.
```

Im Beispiel lehnt der *einseitige* Test zum Regressor `x` (bei einem erwartungskonformen Vorzeichen des Koeffizienten) seine Nullhypothese ab, der *zweiseitige* hingegen nicht:

```
[1] 0.04031867
[1] 0.08063734
```

Manche **R**-Ausgaben lassen sich mit der Funktion **spsspivottable.Display()** aus dem **R**-Paket zur SPSS-Unterstützung, das mit den **R**-Essentials installiert wird, in eine SPSS-Pivot-Tabelle wandeln. Nach einer geringfügigen Erweiterung des letzten **R**-Programms

```
BEGIN PROGRAM R.
library(MASS)
casedata <- spssdata.GetDataFromSPSS()
huber <- rlm(formula = y ~ x, data = casedata)
results <- summary(huber)
spsspivottable.Display(results$coefficients,
                       title = "Koeffizienten",
                       format = formatSpec.GeneralStat)
END PROGRAM.
```

erhalten wir als Ausgabe die folgende Pivot-Tabelle:

Koeffizienten

	Value	Std. Error	t value
(Intercept)	,221	,530	,418
x	,886	,507	1,747

Statistische Auswertungsfunktionen in **R** organisieren ihre Ausgabe in der Regel als Liste mit Komponenten unterschiedlichen Typs. Wir behandeln diesen flexiblen **R**-Datentyp in Abschnitt 5.3.4.6. Um zu

erfahren, welche Komponenten in einer Listenausgabe vorhanden und für die Wandlung in eine Pivot-Tabelle verfügbar sind, kann man die **str()** – Funktion verwenden, z.B.:

```
str(results)
```

Im Beispiel hat sich herausgestellt, dass `results` eine Komponente namens `coefficients` enthält. Diese Komponente wird in obigem `spsspivottable.Display()` - Aufruf als erstes Argument verwendet. Ihm folgen ein frei wählbarer Titel und die Formatangabe `formatSpec.GeneralStat`, die in den meisten Fällen zu einem sinnvollen Ergebnis führt.

4.3 SPSS-Variablen mit R erstellen

Das folgende Programm nach Levesque (2011, S. 347) berechnet für eine SPSS-Variable mit Hilfe der **R**-Funktion **mean()** den Mittelwert aller Fälle und erstellt in SPSS ein um die Mittelwertsvariable erweitertes Datenblatt.

```
DATA LIST FREE /salary.
BEGIN DATA
1
2
3
4
END DATA.

BEGIN PROGRAM R.
dict <- spssdictionary.GetDictionaryFromSPSS()
casedata <- spssdata.GetDataFromSPSS()
varSpec <- c("meansal","Mean Salary","0","F8.2","scale")
dict <- data.frame(dict,varSpec,stringsAsFactors=FALSE)
spssdictionary.SetDictionaryToSPSS("results",dict)
casedata <- data.frame(casedata,mean(casedata$salary))
spssdata.SetDataToSPSS("results",casedata)
spssdictionary.EndDataStep()
END PROGRAM.
```

Mit der **R**-Funktion **spssdictionary.GetDictionaryFromSPSS()** werden die Variablendeklarationen aus der SPSS-Arbeitsdatei in eine **R** - Datentabelle mit dem Namen `dict` übernommen:

```
dict <- spssdictionary.GetDictionaryFromSPSS()
```

Um eine neue Variablendeklaration zu ergänzen, wird zunächst ein Vektor vom Datentyp **character** erstellt und im **R**-Objekt `varSpec` abgelegt:

```
varSpec <- c("meansal","Mean Salary","0","F8.2","scale")
```

Hier müssen 5 SPSS-Variablenattribute in vorgeschriebener Reihenfolge durch eine Zeichenkette festgelegt werden:

- **Variablenname**
  Im Beispiel: `"meansal"`
- **Variablenlabel**
  Bei Verzicht auf ein Label ist eine leere Zeichenkette anzugeben ("").
  Im Beispiel: `"Mean Salary"`
- **Variablentyp**
  Für numerische Variablen ist eine 0 anzugeben, für Zeichenkettenvariablen die Länge (von 1 bis 32767).
  Im Beispiel: `"0"`

- **Anzeigeformat**
  Im Beispiel: `"F8.2"`
- **Messniveau**
  Erlaubte Werte: **nominal**, **ordinal**, **scale**
  Im Beispiel: `"scale"`

Im Beispielprogramm wird der Data Frame `dict` über die zum **R**-Kern gehörige Funktion **data.frame()** (vgl. Abschnitt 5.3.4.7.1) um die Variablendeklaration im **character**-Vektor `varSpec` erweitert:

```
dict <- data.frame(dict,varSpec,stringsAsFactors=FALSE)
```

Eine **R**-Datentabelle zum Speichern der Attribute zu den Variablen eines SPSS-Datenblatts enthält für jede SPSS-Variable einen Vektor mit Werten von Typ **character**. Wir erweitern das Datenlexikon `dict` um die Variablendeklaration im **character**-Vektor `varSpec` und verhindern mit dem Argument **`FALSE`** für das **data.frame()** - Argument **stringsAsFactors**, dass der **character**-Vektor bei der Aufnahme in einen Faktor (im Sinne von **R**) gewandelt wird.[1]

Aus dem Ergebnis erstellt die Funktion **spssdictionary.SetDictionaryToSPSS()**

```
spssdictionary.SetDictionaryToSPSS("results",dict)
```

ein neues SPSS-Datenblatt mit dem Namen `results`.

Die **R**-Datentabelle `casedata` mit den SPSS-Variablen wird im folgenden **data.frame()** – Aufruf um die per **mean()** gebildete Mittelwertsvariable erweitert:[2]

```
casedata <- data.frame(casedata, mean(casedata$salary))
```

Den von der **R**-Funktion **mean()** gelieferten numerischen Vektor der Länge 1 erweitert **R** automatisch durch Wiederholung des vorhandenen Elements auf die passende Länge.

Mit der Funktion **spssdata.SetDataToSPSS()** werden die Variablen im erweiterten Data Frame `casedata` in das eben angelegte SPSS-Datenblatt geschrieben:

```
spssdata.SetDataToSPSS("results",casedata)
```

Am Ende der Datenblatt-Erstellung sollte die Funktion **spssdictionary.EndDataStep()** aufgerufen werden.

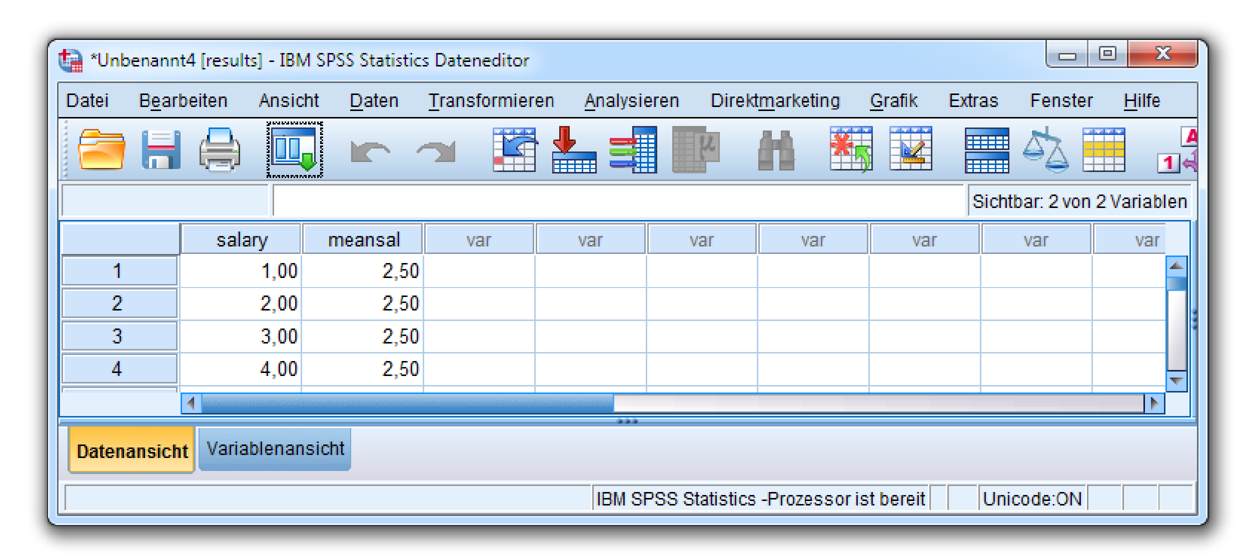
Das Ergebnis:

---

[1] Die voreingestellte Wandlung wäre für die weitere Verarbeitung im konkreten Beispielprogramm kein Problem, doch sollte generell die Struktur einer **R**-Datentabelle zur Aufbewahrung von SPSS-Variablenattributen respektiert werden.

[2] Im Hinblick auf den Abschnitt 5.3.4.7 soll darauf hingewiesen werden, dass im Beispiel auf das explizite Benennen der neuen Variablen in der **R**-Datentabelle verzichtet wird, so dass eine automatische Namensvergabe stattfindet. Das Ergebnis verrät der **R**-Funktionsaufruf `str(casedata)`:

```
'data.frame': 4 obs. of  2 variables:
 $ salary               : num  1 2 3 4
 $ mean.casedata.salary.: num  2.5 2.5 2.5 2.5
```



Mit der Mittelwertsbildung wurde ein möglichst einfaches Beispiel gewählt, um die Erstellung von SPSS-Variablen durch **R** zu demonstrieren. Diese Aufgabe ist mit SPSS-internen Mitteln einfacher zu lösen:

```
AGGREGATE
  /OUTFILE=* MODE=ADDVARIABLES
  /meansal=MEAN(salary).
```

Es kann sich aber leicht die Situation ergeben, dass zur Lösung einer Datentransformationsaufgabe **R**-Syntax in Frage kommt, z.B. weil ein in **R** implementierter Algorithmus übernommen werden soll.

5 R als statistikorientierte Programmierumgebung

Mit zunehmender Komplexität der **R**-Beispiele in den obigen Abschnitten ist vermutlich die Motivation der Leser(innen) gewachsen, sich mit *R als Programmierumgebung für Datenanalyse und Grafik* (dt. Übersetzung des Untertitels von Venables et al. 2014) systematisch zu beschäftigen, um Sicherheit bei der Anwendung zu gewinnen.

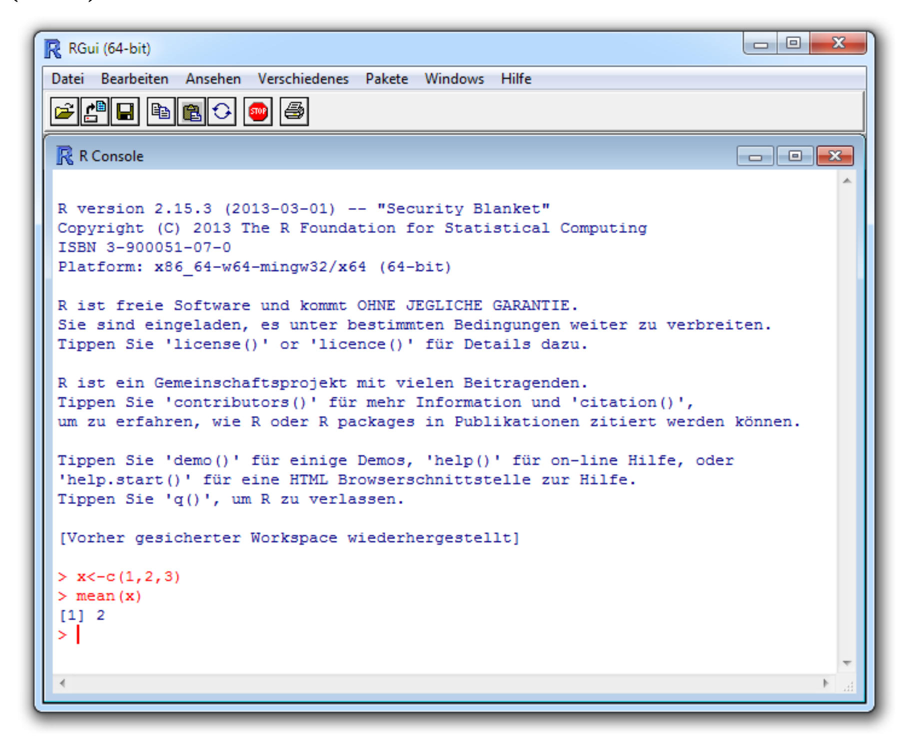
5.1 RGui zur direkten Interaktion mit R

Im aktuellen Abschnitt 5 werden die **R**-Funktionen der Kürze halber in der Regel *ohne* Einrahmung durch die SPSS-Kommandos

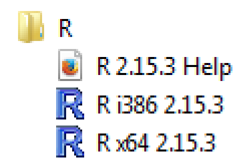
```
BEGIN PROGRAM R.
 . . .
END PROGRAM.
```

gezeigt.

Zum Erlernen der **R**-Syntax ist die direkte Interaktion mit dem **R**-Interpreter über die grafische **R**-Bedienoberfläche (RGui) ohnehin sinnvoller:



Bei einer Windows-Version mit 64-Bit - Architektur steht das RGui als 32- oder 64-Bit - Anwendung bereit. Mittlerweile ist auch die 64-Bit - Version ausgereift und sollte in der Regel bevorzugt werden. Hier ist die Startmenügruppe zu R 2.15.3 zu sehen:

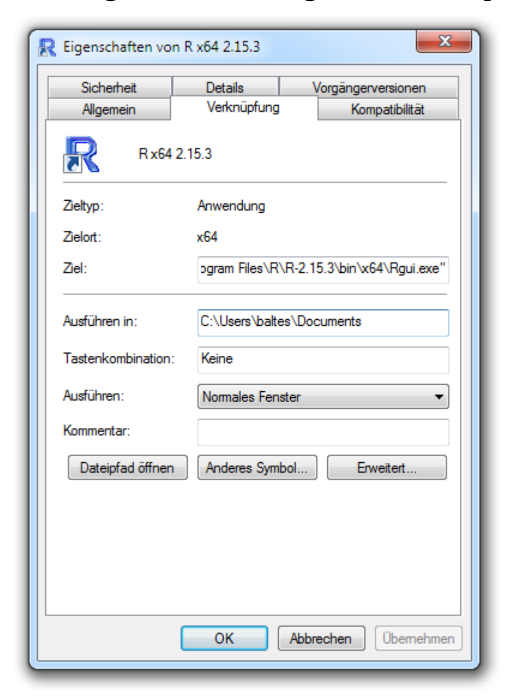


Die im RGui üblichen Farben für Eingaben (rot) und Ausgaben (blau) werden auch im Manuskript verwendet.

Weil im Konsolenfenster der RGui-Bedienoberfläche sämtliche Eingaben und **R**-Antworten protokolliert werden, ist gelegentlich sinnvoll, per Kontextmenü oder mit dem Tastenbefehl **Strg+L** für eine leere **R**-Konsole zu sorgen.

5.1.1 Arbeitsverzeichnis

In einer R-Sitzung dient das **Arbeitsverzeichnis** (engl.: *working directory*) als Voreinstellung beim Lesen und Schreiben von Dateien. Initial ist es identisch mit dem aktuellen Verzeichnis beim Start von **R** (**Startverzeichnis**). Startet man das RGui über seine Verknüpfung im Startmenü, hängt das das Startverzeichnis vom Feld **Ausführen in** des Eigenschaftsdialogs zur Verknüpfung ab, z.B.:



Man kann in der R-Konsole das aktuelle Arbeitsverzeichnis mit der Funktion **getwd()** ermitteln, z.B.:

```
> getwd()
[1] "C:/Users/baltes/Documents"
```

Um das Arbeitsverzeichnis zu wechseln, kann man den RGui-Menübefehl

Datei > Verzeichnis wechseln

oder die **R**-Funktion **setwd()** verwenden, z.B.:

```
> setwd("U:/Eigene Dateien/R")
```

Weil in **R** (wie in vielen anderen Programmiersprachen) der Rückwärts-Schrägstrich als Einleitungszeichen für Escape-Sequenzen reserviert ist (z.B. **\n** für den Zeilenwechsel), muss bei Windows-Pfadangaben ersatzweise ein doppelter Rückwärts-Schrägstrich oder ein Vorwärts-Schrägstrich verwendet werden.

Mit der Funktion **dir()** fordert man eine Liste der Dateisystemobjekte im Arbeitsverzeichnis an:

```
> dir()
```

5.1.2 Workspace und Anweisungsgedächtnis

Die in einer Sitzung erzeugten Objekte (z.B. Vektoren und Datentabellen) landen im so genannten **Workspace**. Mit der Funktion **ls()** kann man sich die Objekte im Workspace auflisten lassen, z.B.:

```
> ls()
[1] "eimer" "x"
```

Der etwas längere und dabei informativere Aliasname für diese Funktion lautet **objects()**.

Über die **apropos()** - Funktion kann man alle Objekte ermitteln, die einen bestimmten Namensbestandteil besitzen. Die folgenden Kommandos

```
> eimer <- c(1, 2, 3)
> apropos("eim")
```

führen zur Ausgabe:

```
[1] ".mergeImportMethods"    "eimer"                  "getNamespaceImports"
[4] "namespaceImport"        "namespaceImportClasses" "namespaceImportFrom"
[7] "namespaceImportMethods" "saveNamespaceImage"
```

Man kann die Funktion **apropos()** im RGui auch per Menübefehl nutzen:

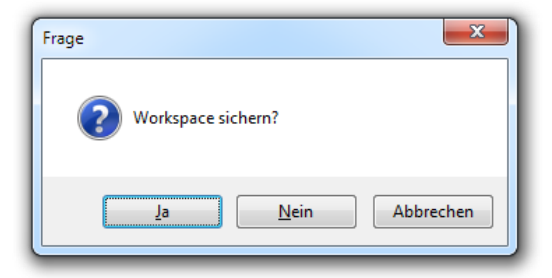
Hilfe > Apropos

Neben den Datenobjekten merkt sich das RGui auch die abgeschickten Anweisungen. Über die vertikalen Pfeiltasten kann man früher verwendete Anweisungen zurückholen.

Das RGui verwendet die folgenden Dateien im Arbeitsverzeichnis, um den Workspace und das Kommandogedächtnis zwischen zwei Sitzungen aufzubewahren:

- **.RData**
  Hier speichert das RGui den Workspace.
- **.Rhistory**
  Hier speichert das RGui das Kommandogedächtnis.

Beim Starten liest **R** den Workspace und das Kommandogedächtnis aus diesen Dateien, und beim Beenden einer Sitzung werden Datenobjekte sowie gespeicherte Anweisungen der Sitzung nach einer zustimmenden Antwort auf die folgende Frage



in die beiden Dateien gesichert.

Um *alle* Datenobjekte der aktuellen Sitzung in eine *wählbare* Datei mit der Namenserweiterung **.RData** zu sichern bzw. von dort zu laden, kann man die folgenden Menübefehle

- **Datei > Sichere Workspace** bzw.
- **Datei > Lade Workspace**

verwenden, oder mit Kommandos arbeiten, z.B.:

- `> save.image("ws.RData")`
- `> load("ws.RData")`

Vor dem Speichern aller Objekte kann es sinnvoll sein, mit der Funktion **rm()** überflüssige Objekt aus dem Workspace zu entfernen, z.B.:

```
> rm(v1, v2, matz)
```

Um das Kommandogedächtnis der aktuellen Sitzung in eine wählbare Datei zu sichern bzw. von dort zu laden, kann man die Menübefehle

- **Datei > Speichere History** bzw.
- **Datei > Lade History**

verwenden, oder mit Kommandos arbeiten, z.B.:

- `> savehistory("komm.Rhistory")`
- `> loadhistory("komm.Rhistory")`

Alle per **save.image()** oder **savehistory()** ohne Pfadangabe geschriebenen Dateien landen im Arbeitsverzeichnis. Wie man es ermittelt oder verändert, wurde im Abschnitt 5.1.1 erläutert.

5.1.3 Sichern und Laden einzelner Datenobjekte im Binärformat von R

Eine **RData**-Datei kann in **R** als binärformatige Datendatei analog zu einer **SAV**-Datei in SPSS genutzt werden. Oft ist es sinnvoll, ein wichtiges Datenobjekt (z.B. eine Datentabelle mit den Variablen einer Studie) in einer eigenen Datei zu speichern. Dazu eignet sich die Funktion **save()**, z.B.:

```
> save(Daten, file="Daten.RData")
```

Mit der Funktion **load()** befördert man das in einer Datei befindliche Datenobjekt in den Workspace einer späteren Sitzung:

```
> load("Daten.RData")
```

Eine bequeme Möglichkeit zum Öffnen einer **RData**-Datei besteht darin, die Datei per Drag & Drop vom Fenster des Windows-Explorers auf das Fenster der **R**-Konsole zu bewegen.

5.1.4 Konfigurationsoptionen

Diverse Verhaltensmerkmale von **R** lassen sich über die Funktion **options()** beeinflussen. Ein Funktionsaufruf ohne Argumente hat eine (längliche) Auflistung sämtlicher Optionen zur Folge. Übergibt man eine Option als Zeichenfolgen-Argument, erfährt man den aktuellen Wert, z.B.:

```
> options("digits")
$digits
[1] 7
```

Um die meisten Optionen muss man sich wegen sinnvoller Voreinstellungen nicht kümmern. Anschließend werden Optionen vorgestellt, bei denen sich eine Änderung lohnen könnte. Soll eine Einstellungsänderung bei jedem **R**-Start gültig sein, fügt man den zugehörigen **options()**-Aufruf in eine Initialisierungsdatei ein (siehe Abschnitt 5.1.5).

Genauigkeit der Ergebnisausgabe

Per Voreinstellung werden Berechnungsergebnisse mit 7 Stellen Genauigkeit angezeigt, z.B.:

```
> log(2)
[1] 0.6931472
```

Über das Argument **digits** der Funktion **options()** lässt sich eine alternative Anzeigegenauigkeit einstellen, z.B.:

```
> options(digits=15)
> log(2)
[1] 0.693147180559945
```

Automatisch geladene Pakete
Alle **R**-Funktionen befinden sich in Paketen und können erst nach dem Laden ihres Pakets genutzt werden (siehe Abschnitt 5.2). Über die Option **defaultPackages** legt man fest, welche Pakete automatisch geladen werden sollen. Hier ist die Voreinstellung für die Option zu sehen:

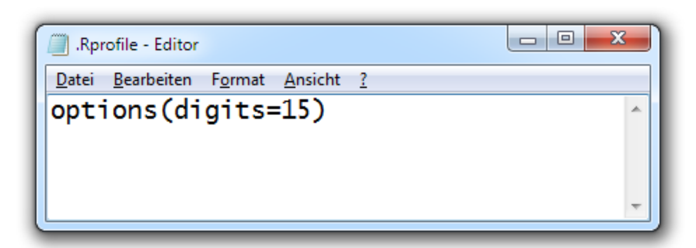
```
> options("defaultPackages")
$defaultPackages
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"
```

5.1.5 Initialisierungsdateien

Als Initialisierungsdateien mit Anweisungen, die beim Programmstart ausgeführt werden sollen, fungieren:

- **Rprofile.site** im **etc**-Unterordner des **R**-Programmordners (**C:\Program Files\R\R-2.15.3\etc**)
- **.Rprofile** im Startverzeichnis (mit einem Punkt als erstem Zeichen im Dateinamen)

Mit der folgenden Datei **.Rprofile** im Startverzeichnis wird eine Ausgabegenauigkeit von 15 Dezimalstellen angeordnet:



5.2 Pakete

R-Funktionen befinden sich in Paketen, die geladen sein müssen, um die darin enthaltenen Funktionen nutzen zu können. Welche Pakete aktuell geladen sind, erfährt man durch einen Aufruf der Funktion **.packages()**, z.B.:

```
> (.packages())
[1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[7] "base"
```

In der etwas gewöhnungsbedürftigen Syntax sorgen die runden Klammern um den Funktionsaufruf dafür, dass seine Rückgabe (ein Vektor mit Elementen vom Typ **character**) ausgegeben wird. Einige Pakete (darunter stets das Paket **base**) werden automatisch geladen. Welche Pakete neben **base** automatisch geladen werden, kann man über die Option **defaultPackages** erfahren und festlegen (vgl. Abschnitt 5.1.4).

Alternativ zu **.packages()** kann man die Funktion **search()** dazu benutzen, die geladenen Pakete aufzulisten, z.B.:

```
> search()
[1] ".GlobalEnv"         "package:stats"      "package:graphics"
[4] "package:grDevices" "package:utils"      "package:datasets"
[7] "package:methods"    "Autoloads"          "package:base"
```

Die eigentliche Aufgabe der Funktion **search()** besteht darin, den Suchpfad für Objekte der **R**-Sitzung aufzulisten (siehe Abschnitt 5.3.3.2). Im RGui unter Windows lässt sich der **search()** - Aufruf auch mit dem folgenden Menübefehl auslösen:

Verschiedenes > Liste Suchpfad auf

5.2.1 Pakete laden

Zum Laden eines Pakets taugt die Funktion **library()**, z.B.:

```
> library(MASS)
```

Ein Funktionsaufruf ohne Argumente

```
> library()
```

führt zu einer Liste mit allen installierten Paketen ...

- in der installations-allgemeinen Bibliothek, z.B.: **C:\Program Files\R\R-2.15.3\library**
- und in der persönlichen Bibliothek, z.B. **C:\Users\baltes\Documents\R\win-library\2.15**

Über das Argument **lib.loc** ist es möglich, ein Paket aus einen beliebigen Ordner zu laden, z.B.:

```
> library(mice, lib.loc="E:\\Daten\\R\\library\\2.15")
```

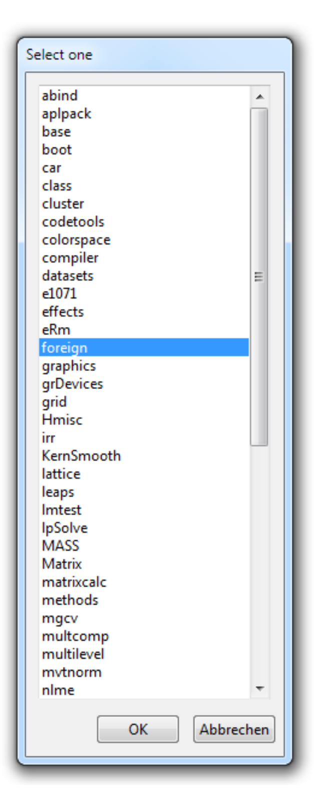
Beinhalten mehrere geladene Pakete namensgleiche Funktionen, gewinnt das zuletzt geladene Paket. Über den Operator **::** ist es aber möglich, die Funktion aus einem bestimmten Paket anzusprechen, z.B.

```
> mypack::func()
```

Um ein Paket *per Dialogbox* zu laden, kann man im RGui den Menübefehl

Pakete > Lade Paket

verwenden, aus der Liste mit allen installierten Paketen (siehe oben) ein Exemplar wählen und mit **OK** quittieren:



In den meisten Fällen ist jedoch das Laden von Paketen per **library()** - Funktion sinnvoller (z.B. im Rahmen von Skripten).

Zum Laden von Paketen per Syntax taugt auch die Funktion **require()**, die im Vergleich zu **library()** folgende Vorteile besitzt:

- Scheitert das Laden, produziert **library()** eine Fehlermeldung, **require()** hingegen eine Warnung. Beim Aufruf innerhalb einer Funktion führt eine Fehlermeldung zum Abbruch der Funktion, eine Warnung hingegen nicht.
- **require()** liefert eine boolesche Rückgabe zum Erfolg des Aufrufs und kann daher gut in eine bedingte Anweisung integriert werden (vgl. Abschnitt 5.3.8).

5.2.2 Pakete installieren

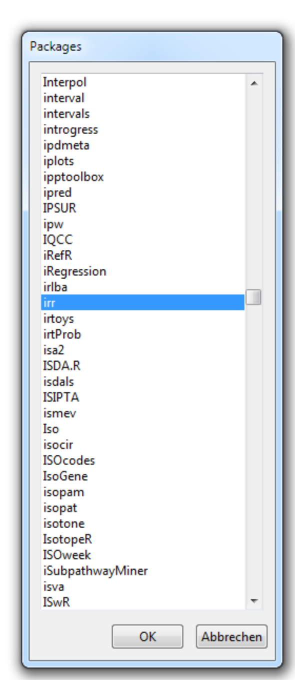
Bei der Arbeit mit **R** ist es häufig erforderlich, Zusatzpakete zu installieren, die bestimmte Auswertungsverfahren implementieren. Wir wollen in Abschnitt 8.3.1 das (in SPSS Statistics nicht verfügbare) **Fleiss-Kappa** zur Beurteilung der Inter-Rater-Übereinstimmung bei mehr als 2 Diagnostikern berechnen lassen und benötigen dazu das **R**-Paket **irr** (*Inter Rater Reliability*), das nicht zum Standardumfang einer **R**-Installation gehört.

Bei bestehender Internet-Verbindung kann die Ergänzungsinstallation bequem im RGui über den folgenden Menübefehl gestartet werden:

Pakete > Installiere Paket(e)

Es erscheint ein Dialog zur Wahl eines Spiegel-Servers zum *Comprehensive R Archive Network* (CRAN) (links). Danach wählt man das gewünschte Paket (rechts):





Wer lieber mit Kommandos arbeitet, wählt zur Installation eines Paketes die Funktion **install.packages()**, z.B.:

```
> install.packages("irr")
```

Auch bei diesem Einstsieg fragt das RGui nach dem bevorzugten Spiegel-Server.

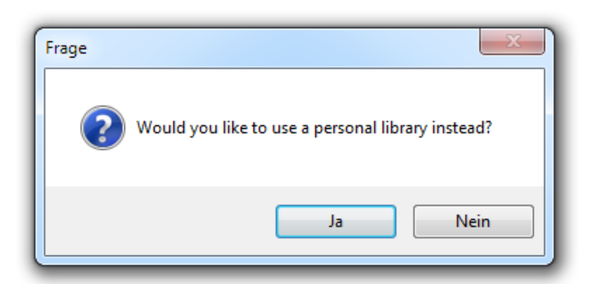
Ist auf dem Zielrechner keine Internet-Verbindung vorhanden, kann man die benötigten Pakete auf einem lokalen Datenträger als ZIP-Dateien bereithalten und über den folgenden Menübefehl installieren:

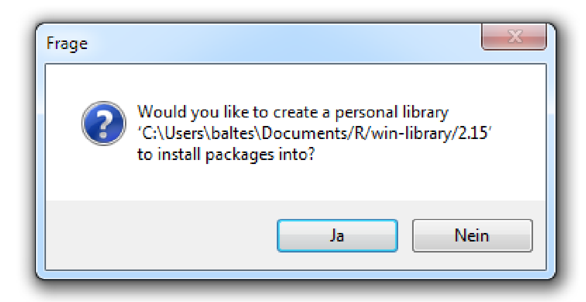
Pakete > Installiere Paket(e) aus lokalen Zip-Dateien

Sofern entsprechende Schreibrechte bestehen, werden neue Pakete im **library** - Unterordner des **R**-Installationsordners abgelegt, und die Pakete stehen allen Benutzern zur Verfügung. Bei einer Installation von R 2.15.3 unter Windows 7 landen die Pakete per Voreinstellung hier:

C:\Program Files\R\R-2.15.3\library

Bestehen bei der Installation eines Zusatzpakets keine Schreibrechte für diesen Ordner, wird die Einrichtung einer **persönlichen Bibliothek** vorgeschlagen:





Vor der geplanten Installation von Paketen, die in der allgemeinen Bibliothek (im Programmordner) landen und damit für alle Benutzer verfügbar sein sollen, muss **R** mit administrativen Rechten gestartet werden, damit ein Schreibzugriff auf den Programmordner möglich ist. Sobald ein persönlicher Bibliotheksordner vorhanden ist, wird dieser jedoch vorgezogen, wie die Ausgabe der Funktion **.libPaths()** zeigt:

```
> .libPaths()
[1] "C:/Users/baltes/Documents/R/win-library/2.15"
[2] "C:/Program Files/R/R-2.15.3/library"
```

In dieser Lage kann man zur Installation die Funktion **install.packages()** benutzen und über das Argument **lib** den Installationsort bestimmen, z.B.:

```
> install.packages("irr", lib="C:/Program Files/R/R-2.15.3/library")
```

Eine weitere Möglichkeit, die Paketinstallation in der allgemeinen Bibliothek zu erzwingen, besteht darin, den persönlichen Bibliotheksordner vorübergehend (vor dem Start von **R**) durch Umbenennen zu deaktivieren.

Bei der automatisch ablaufenden Installation werden auch Abhängigkeiten von anderen Paketen berücksichtigt, wie die Konsolen-Protokollausgabe zum Beispiel zeigt:

```
--- Bitte einen CRAN Spiegel für diese Sitzung auswählen ---
Warnung in install.packages(NULL, .libPaths()[1L], dependencies = NA, type = type) :
  'lib = "C:/Program Files/R/R-2.15.3/library" ist nicht schreibbar
also installing the dependency ‘lpSolve’

versuche URL 'http://ftp5.gwdg.de/pub/misc/cran/bin/windows/contrib/2.15/lpSolve_5.6.7.zip'
Content type 'application/zip' length 678055 bytes (662 Kb)
URL geöffnet
downloaded 662 Kb

versuche URL 'http://ftp5.gwdg.de/pub/misc/cran/bin/windows/contrib/2.15/irr_0.84.zip'
Content type 'application/zip' length 94601 bytes (92 Kb)
URL geöffnet
downloaded 92 Kb

Paket ‘lpSolve’ erfolgreich ausgepackt und MD5 Summen abgeglichen
Paket ‘irr’ erfolgreich ausgepackt und MD5 Summen abgeglichen
```

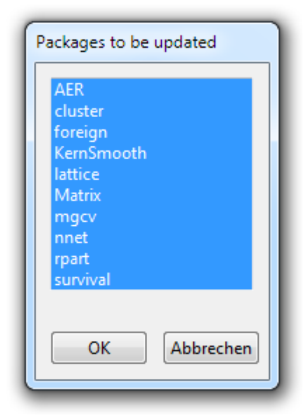
Weil **R** bei fehlenden Schreibrechten im Programmordner geschickt ausweicht, können Benutzer z.B. auf einem ZIMK-Pool-PC persönlich benötigte Pakete problemlos nachrüsten. Das gilt natürlich auch für Pakete, die von SPSS aus genutzt werden sollen, wobei die Installation aber über das RGui geschehen muss.

5.2.3 Installierte Pakete aktualisieren

Vor einer geplanten Aktualisierung der installierten Pakete muss **R** mit administrativen Rechten gestartet werden, damit ein Schreibzugriff auf das Programmverzeichnis möglich ist. Um die Aktualität der installierten Pakete zu prüfen und nötigenfalls Updates zu installieren, bietet das RGui den Menübefehl

Pakete > Aktualisiere Pakete

Nachdem die Liste der betroffenen Pakete mit **OK** quittiert worden ist, läuft die Aktualisierung automatisch ab:



Die Aktualisierung aller Pakete lässt sich auch über die Funktion **update.packages()** anfordern:

```
> update.packages()
```

5.2.4 Task Views

Um das Installieren und Aktualisieren von **R**-Paketen zu erleichtern, wurden sogenannte **Task Views** definiert, die aus einer mehr oder weniger großen Zahl von Paketen bestehen und komplett mit

```
> install.views("name")
```

installiert sowie mit

```
> update.views("name")
```

aktualisiert werden können. Eine Beschreibung der verfügbaren Paketbündel findet sich hier:

http://cran.r-project.org/web/views/

Um Task Views nutzen zu können, muss zunächst das **R**-Paket **ctv** (*CRAN Task Views*) installiert werden.

5.2.5 Pakete entladen

Ein zuvor per **library()** geladenes Paket kann per **detach()** wieder entladen werden, z.B.:

```
> detach(package:MASS)
```

5.2.6 Pakete zitieren

Wenn eine Veröffentlichung auf **R**-Paketen basiert, sollte der Urheber aus Respekt vor seiner geistigen Leistung und zur Orientierung des Lesers angegeben werden. Mit der Funktion **citation()** erfährt man, wie ein **R**-Paket zitiert werden muss, z.B.:

```
> citation("MASS")
```

Im Beispiel stellen sich mit W.N. Venables und B. D. Ripley zwei herausregende Förderer der **R**-Entwicklung als Autoren heraus:

```
To cite the MASS package in publications use:

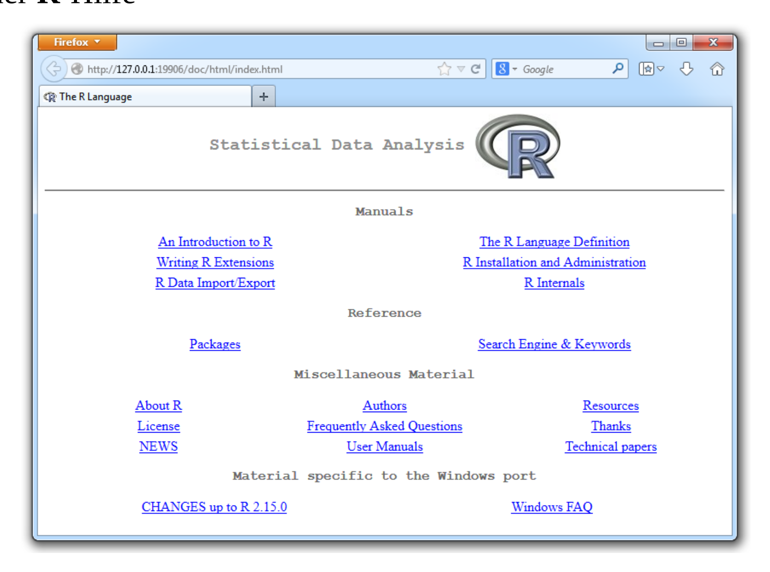
  Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth
  Edition. Springer, New York. ISBN 0-387-95457-0
```

5.3 Elementare Eigenschaften der Programmiersprache R

5.3.1 Hilfe und Dokumentation

5.3.1.1 Hilfe aufrufen

Die HTML-Startseite der **R**-Hilfe



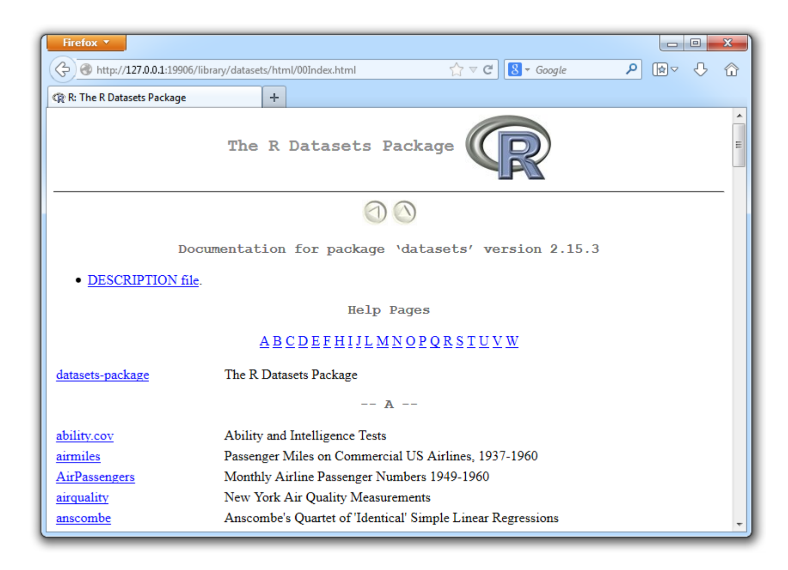
ist erreichbar über das Kommando

```
> help.start()
```

oder den RGui-Menübefehl:

Hilfe > HTML-Hilfe

Über den Link **Packages** gelangt man zu einer Liste mit allen installierten Paketen und über einen Mausklick auf ein Paket erreicht man die zugehörige Hilfeseite, z.B. beim Paket **datasets**:



Informationen zu einem Schlüsselwort der Programmiersprache **R** oder zu einer **R**-Funktion aus einem *geladenen* Paket erhält man über die Funktion **help()**, z.B.:

```
> help("mean")
```

Kurzform:

```
> ?"mean"
```

Bei vielen Suchbegriffen kann die Umrahmung durch Anführungszeichen entfallen, z.B.:

```
> help(mean)
> ?mean
```

Bei Schlüsselwörtern der Programmiersprache **R** ist sie jedoch erforderlich, z.B.:

```
> ?"if"
```

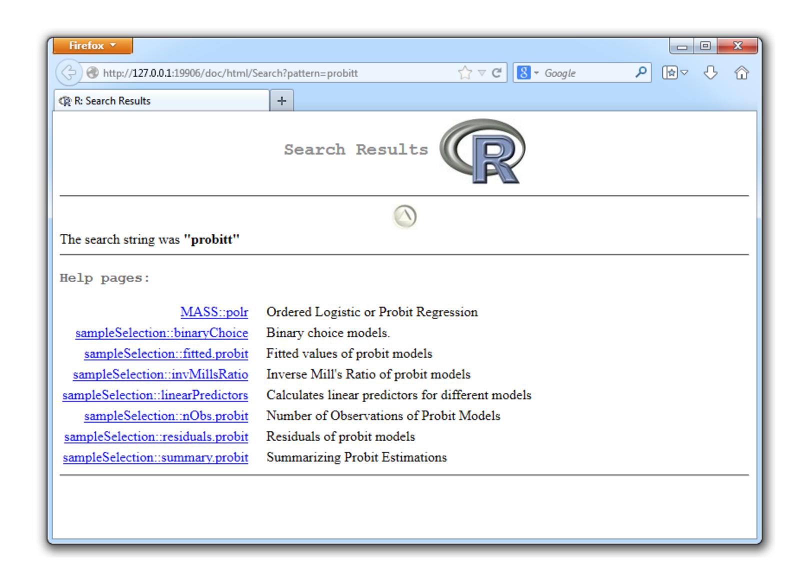
Um die Hilfedateien (auch von nicht geladenen Paketen) *fehlertolerant* nach einem Begriff zu durchsuchen, eignet sich die die Funktion **help.search()**, z.B.:

```
> help.search(probitt)
```

Kurzform (mit demselben Tippfehler):

```
> ??probitt
```

Im Beispiel stellt sich heraus, dass z.B. die Dokumentation zur Funktion **polr** im Paket **MASS** einen ähnlich geschriebenen Begriff enthält:



5.3.1.2 Beispiele aus den Hilfetexten ausführen lassen

In den Hilfetexten zu den **R**-Funktionen finden sich regelmäßig Beispiele, etwa zur **mean**-Funktion:

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

Über die Funktion **example()** kann man diese Beispiele in Aktion erleben. Durch den folgenden Aufruf

```
> example("mean")
```

erhält man das Ergebnis:

```
mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

5.3.1.3 Elektronische Handbücher

Das Hilfe-Menü im RGui bietet über

Hilfe > Handbücher (PDF)

zahlreiche Handbücher im PDF-Format an, z.B.:

- *An Introduction to R* (Venables et al. 2014).
- *R Language Definition* (R Development Core Team 2014)

Auf der CRAN-Webseite (*Comprehensive R Archive Network*)

http://cran.r-project.org/

finden sich unter der Überschrift **Documentation** diverse Handbücher zu **R**, die vom Kern-Team und aus anderen Quellen stammen.

5.3.2 Bezeichner und Kommentare

Für die Namen von Variablen und Funktionen in **R** gelten folgende Regeln:

- Erlaubte Zeichen:
  - Buchstaben (inkl. dt. Umlaute) und Ziffern
  - Punkt (.) und Unterstrich (_)
- Das erste Zeichen muss ein Buchstabe oder Punkt sein, wobei Bezeichner mit führendem Punkt für spezielle Zwecke reserviert bleiben sollten.
- Die Länge ist beliebig.
- Die **Groß-/Kleinschreibung ist relevant**.
- Schlüsselwörter der Programmiersprache **R** (z.B. **if**, **TRUE**) scheiden als Bezeichner aus. Eine Liste der reservierten Wörter erhält man mit:
  ```
  > ?Reserved
  ```
- Die Namen müssen in ihrer Umgebung (z.B. in ihrem Paket) eindeutig sein.

In der empirischen Forschungspraxis lohnt es sich, einigen Aufwand von Zeit und Phantasie in die Bildung von Variablennamen zu stecken, die einerseits kurz und andererseits informativ sein sollten. Auf der folgenden Webseite

http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml

finden sich die folgenden formalen Kriterien:

- Es sollten ausschließlich Kleinbuchstaben verwendet werden.
  Dieses Kriterium zu verletzen und Variablennamen mit einem Großbuchstaben beginnen zu lassen, hat allerdings bei graphischen Darstellungen die angenehme Konsequenz, dass spontan perfekte Beschriftungen erscheinen (z.B. der Achsentitel *Gewicht* statt *gewicht*).
- Bei zusammengesetzten Bezeichnungen sollten die Wörter durch einen Punkt getrennt werden, z.B.: `buecher.gelesen`.
- Alternativ ist bei zusammengesetzten Bezeichnungen das sogenannte *Camel-Casing* erlaubt, z.B.: `buecherGelesen`, womit an speziellen Positionen doch Großbuchstaben toleriert werden.
- Unterstriche zur Trennung von Wörtern sind unerwünscht, z.B.: ~~`buecher_gelesen`~~.

Für numerische Literale (Zahlen als Bestandteile von **R**-Anweisungen) ist als **Dezimaltrennzeichen** der **Punkt** zu verwenden, z.B.:

```
> 2*3.1415926
[1] 6.283185
```

Speziell bei gespeicherten Sequenzen von **R**-Anweisungen, die als *Skripte* bezeichnet werden (vgl. Abschnitt 5.4), sind **Kommentare** hilfreich für der Verwendung durch andere Personen und für die spätere Nutzung durch den Autor. Mit dem Doppelkreuz wird der Zeilenrest als Kommentar gekennzeichnet, z.B.:

```
sq <- sum(a^2) # Summe der quadrierten Elemente im Vektor a
```

5.3.3 Funktionen

R besitzt eine große Zahl von eingebauten Funktionen, die sich durch Zusatzpakete oder eigene Definitionen noch beliebig erweitern lässt. Damit kann man ...

- mathematische und insbesondere auch statistische Operationen anfordern
- Diagramme erstellen
- Datenverarbeitungsaufgaben erledigen

R betrachtet Funktionen als modifizierbare Datenstrukturen. Eine **R**-Funktion kann andere Funktionen verändern oder auch neu erstellen. Details finden sich in der **R**-Sprachbeschreibung (R Development Core Team 2014).

5.3.3.1 Regeln für den Aufruf von Funktionen

Die Argumente von Funktionsaufrufen werden als **Wertargumente** behandelt, d.h. die Funktion erhält eine Kopie des Arguments, und funktionsintern vorgenommenen Veränderungen bleiben ohne Effekt auf den Aufrufer.

Beim Funktionsaufruf sind **Positions- und Namensargumente** möglich:

- Wird die Argumentreihenfolge der Definition eingehalten, kann man die Werte ohne Namen hintereinander durch Kommata getrennt angeben.
- Paare aus dem Argumentnamen, einem Gleichheitszeichen und dem zugehörigen Wert dürfen in beliebiger Reihenfolge auftreten. Man darf die Argumentnamen abkürzen, solange Eindeutigkeit besteht, sollte aber dabei auf die Lesbarkeit achten.
- Namentlich bedachte Argumente gelten als versorgt und werden nicht mehr berücksichtigt, wenn die unbenannten Werte des Funktionsaufrufs in der Reihenfolge ihres Auftretens den Argumenten aus der Funktionsdefinition zugeordnet werden.

Besitzt ein Argument eine Voreinstellung, muss beim Aufruf kein Wert angegeben werden.

Eine Besonderheit von **R** ist das **Dreipunktargument** (**…**):

- Es kann für eine Serie von Argumenten stehen, z.B. in der bereits mehrfach erwähnten und benutzten Verknüpfungsfunktion **c()**, die eine beliebigen Anzahl von Elementen entgegennimmt und daraus einen Vektor erstellt:

  c(..., recursive = FALSE)
- Damit kann eine Funktion Argumente entgegennehmen, die sie an eine intern aufgerufene Funktion weiterreicht.

Besitzt eine Funktion keine Argumente, ist beim Aufruf trotzdem eine leere Parameterliste (über ein Paar runder Klammern) an den Namen anzuhängen.

5.3.3.2 Elementare Funktionen und Zuweisungsoperator

Die anschließend vorgestellten **R**-Funktionen werden häufig benötigt und sind generell (ohne vorheriges Laden von Paketen) verfügbar.

Verkettungsfunktion c() und Zuweisungsoperator

Die Verkettungsfunktion **c()** (*combine*, *concatenate*) erzeugt einen Vektor (vgl. Abschnitt 5.3.4.2) bestehend aus den durch Kommata getrennten Argumenten, z.B.:

```
> x <- c(1, 2, 3)
```

Im Beispiel wird das Ergebnis der Variablen `x` zugewiesen, wobei der aus den beiden Zeichen „**<-**“ bestehende **Zuweisungsoperator** zum Einsatz kommt, den wir vor seiner „offiziellen“ Behandlung (siehe Abschnitt 5.3.7.6) in vielen Beispielen benötigen werden.

Als Argumente der Verkettungsfunktion sind auch Vektoren erlaubt, so dass man z.B. bequem einen Vektor verlängern kann:

```
> x <- c(x, 4, 5)
```

Mit Hilfe des Zuweisungsoperators lässt sich auch ein einzelner Wert, der in **R** als einelementiger Vektor aufgefasst wird, in eine Variable schreiben.

```
> k <- 13
```

Wertausgabe mit print()
Mit der Funktion **print()** gibt man den Inhalt einer Variablen aus, z.B.:

```
> print(x)
[1] 1 2 3 4 5
```

Weil dies sehr oft erforderlich ist, kann man die **print()** - Funktion beim interaktiven Arbeiten implizit aufrufen, indem man einen Variablennamen per Kommandozeile abschickt, z.B.:

```
> x
[1] 1 2 3 4 5
```

Der implizite **print()** - Aufruf klappt auch bei der **R**-Nutzung über ein SPSS-Syntaxfenster. Er klappt hingegen nicht ...

- in der eingebetteten Anweisung einer **for**-Schleife (vgl. Abschnitt 5.3.8.3)
- in einem per **source()** ausgeführten Skript (vgl. Abschnitt 5.4)
- in einer selbst definierten **R**-Funktion (vgl. Abschnitt 5.6)

Am Zeilenanfang einer **print()** - Ausgabe erscheint zwischen eckigen Klammern die Indexnummer des ersten Ergebniswerts in der jeweiligen Zeile, was nur bei einer mehrzeiligen Ergebnisausgabe von Bedeutung ist. Zwecks Demonstration verwenden wir im Vorgriff den Sequenzoperator (vgl. Abschnitt 5.3.7.4):[1]

```
> y <- 1:30
> y
[1]   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30
```

Um den Wert einer Zuweisung ohne expliziten **print()** - Aufruf ausgeben zu lassen, setzt man runde Klammern um die Zuweisung, z.B.:

```
> (x <- c(1, 2, 3, 4, 5))
[1] 1 2 3 4 5
```

length(x)
Die Funktion **length()** liefert die Anzahl der Elemente in einem Objekt (z.B. in einem Vektor oder in einer Liste), z.B.:

```
> length(x)
[1] 5
```

Man kann der Länge eine positive Ganzzahl zuweisen und so am Ende des Vektors Elemente ergänzen oder löschen. Auf diese Weise angehängte Elemente haben den Wert **NA** (*Not Available*), z.B.:

```
> length(x) <- 7
> x
[1]  1  2  3  4  5 NA NA
> length(x) <- 3
> x
[1]  1  2  3
```

[1] Anschließend vergessen wir den Sequenzoperator wieder bis zu seiner offiziellen Vorstellung.

ls()
Diese Funktion listet die Objekte im Workspace auf, z.B.:

```
> ls()
[1] "eimer" "x"
```

Im RGui unter Windows kann man den **ls()** - Aufruf auch mit dem folgenden Menübefehl auslösen:

Verschiedenes > Liste Objekte auf

rm()
Mit der Funktion **rm()** kann man ein Objekt aus dem Workspace entfernen, z.B.:

```
> x
[1] 1 2 3 4 5
> rm(x)
> x
Fehler: Objekt 'x' nicht gefunden
```

Mit dem folgenden Funktionsaufruf, der die **ls()** - Ausgabe als Wert für das Argument **list** verwendet, werden alle Objekte im Workspace gelöscht (ohne Warnung):

```
> rm(list = ls())
```

Im RGui unter Windows kann man den Aufruf auch mit dem folgenden Menübefehl auslösen:

Verschiedenes > Entferne alle Objekte

search(),
Dieser Funktion listet den Suchpfad für Objekte auf, z.B.:

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Er startet mit der globalen Umgebung, welche die Workspace-Objekte enthält, und endet mit dem Paket **base**.

quit() bzw. **q()**
Mit dieser Funktion wird **R** beendet, z.B.:

```
> q()
```

5.3.4 Datentypen

Während SPSS mit der rechteckigen Datenmatrix als der einzigen Datenstruktur auskommt, bietet **R** *mehrere* Datentypen an. Zusammen mit einem Datentyp werden anschließend auch die bei seiner Verwendung häufig benötigten Funktionen vorgestellt. Statt von *Datentypen* spricht die **R**-Literatur auch von *Klassen*. Dementsprechend werden Variablen oft als *Objekte* bezeichnet.

5.3.4.1 Datentypbezogene Funktionen

Mit der **class()**-Funktion befragt, nennt ein Objekt seine Klasse, z.B.:

```
> ma <- matrix(c(1,2,3,4),2)
> class(ma)
[1] "matrix"
```

```
> v <- c(1,2)
> class(v)
[1] "numeric"
```

Befragte Vektor-Objekte nennen den Typ ihrer Elemente (siehe zweites Beispiel). Bei numerischen Vektoren erscheint statt **numeric** in Abhängigkeit von der Erstellungsmethode eventuell eine genauere Typangabe, z.B.:

```
> v <- 1:2
> class(v)
[1] "integer"
```

Bei Objekten mit typidentischen Elementen (Vektor, Faktor, Matrix, Array), meldet die Funktion **mode()** den Elementtyp, z.B.:

```
> mode(mat)
[1] "numeric"
```

Über die Funktion **is.*typ*()** mit booleschem Rückgabewert stellt man für ein Objekt fest, ob es den genannten Datentyp besitzt, z.B.:

```
> is.matrix(mat)
[1] TRUE
> is.vector(mat)
[1] FALSE
```

Über die Funktion **as.*typ*()** lässt sich eine Typumwandlung erzwingen, z.B.:

```
> v <- c(1,2)
> mode(v)
[1] "numeric"
> v <- as.character(v)
> mode(v)
[1] "character"
```

R verwendet eine implizite und dynamische Typisierung von Variablen. Der Datentyp einer Variablen wird also implizit festgelegt und kann später wieder geändert werden, z.B.:

```
> y <- 3
> mode(y)
[1] "numeric"
> y <- 'a'
> mode(y)
[1] "character"
```

5.3.4.2 Vektor

Ein Vektor ist ein Objekt, das eine geordnete Anzahl von Elementen desselben Grundtyps enthält. Man kann den Vektor als den elementarsten und wichtigsten Datentyp in **R** betrachten, weil fast alle anderen Datentypen intern als Vektoren mit speziellen Eigenschaften realisiert sind (Ligges 2007, S. 33).

R kennt keine Skalare und behandelt z.B. eine einzelne Zahl als einen Vektor der Länge 1.

5.3.4.2.1 Vektoren erstellen

Wir haben bereits in zahlreichen Beispielen die Verkettungsfunktion **c()** dazu verwendet, einen Vektor zu erstellen:

```
> x <- c(1, 2, 5)
```

Über eine Funktion mit dem Namen des Elementtyps und einem Argument zur Längenbestimmung erhält man einen Vektor mit Nullinitialisierung der Elemente. Im folgenden Beispiel entsteht ein numerischer Vektor mit 8 Nullen:

```
> n8 <- numeric(8)
> n8
[1] 0 0 0 0 0 0 0 0
```

Mit der **character()** - Funktion erhält man einen Vektor mit leeren Zeichenfolgen als Elementen, z.B.:

```
> c3 <- character(3)
> c3
[1] "" "" ""
```

Mit dem durch einen Doppelpunkt bezeichneten **Sequenzoperator** lässt sich ein Vektor aus Zahlen produzieren, beginnend mit dem linken Operanden und dann im Einserabstand wachsend bis zur letzten Zahl, die den rechten Operanden nicht übertrifft, z.B.:

```
> (x <- 1:5)
[1] 1 2 3 4 5
```

Um eine Sequenz mit alternativer (auch negativer) Schrittweite zu erzeugen, verwendet man die **seq()**-Funktion, z.B.:

```
> (x <- seq(0.3, 0.36, by=0.01))
[1] 0.30 0.31 0.32 0.33 0.34 0.35 0.36
```

5.3.4.2.2 Vektorelemente ansprechen

Ein Element eines Vektors lässt sich per **[ ]** – Operator über einen (1-basierten) Indexwert ansprechen, z.B.:

```
> x <- c(1, 2, 5)
> x[2]
[1] 2
```

Ein versuchter Zugriff auf eine nicht vorhandene Indexposition liefert den Ersatzwert **NA** (*Not Available*), z.B.:

```
x[7]
[1] NA
```

In Abschnitt 5.3.6 werden weitere Optionen zum Indexzugriff behandelt.

5.3.4.2.3 Automatische Typanpassung

Werden Elemente mit verschiedenen Typen in einen Vektor eingefügt, findet eine Anpassung zum allgemeinsten Typ statt, z.B.:

```
> weck <- 1:3
> mode(weck)
[1] "numneric"
> weck <- c(weck, "a")
> mode(weck)
[1] "character"
> weck
[1] "1" "2" "3" "a"
```

5.3.4.2.4 Vektoren mit Elementtyp character oder logical

Als Elemente vom Typ **character** sind einzelne Zeichen und Zeichenfolgen erlaubt, wobei zur Begrenzung alternativ einfache oder doppelte Anführungszeichen verwendet werden dürfen, z.B.:

```
> s <- c("Otto's Welt", 'ist simpel')
> s
[1] "Otto's Welt" "ist simple"
```

Hinter den Werten vom Typ **logical** (**TRUE**, **FALSE**) stecken intern die Zahlen 1 und 0, z.B.:

```
> cond <- TRUE
> mode(cond)
[1] "logical"
> cond+4
[1] 5
```

Es ist erlaubt, aber *nicht* empfehlenswert, die Wahrheitswerte durch ihre Anfangsbuchstaben abzukürzen, weil diese Symbole (**T**, **F**) umdefiniert worden sein könnten, z.B.:

```
> cond <- T
> mode(cond)
[1] "logical"
> T <- 4
> cond <- T
> mode(cond)
[1] "numeric"
```

5.3.4.2.5 Vektoren mit Datumsangaben

Um Vektoren mit Datumsangaben zu erzeugen, die z.B. eine Differenzberechnung zulassen, kann man so vorgehen:

- Per Verkettungsoperator einen Zeichenkettenvektor mit Werten im Format „jjjj-mm-tt“ erstellen
- und die Funktion **as.Date()** auf diesen Vektor anwenden.

Beispiel:

```
> geboren <- as.Date(c("1978-12-31", "1988-07-24", "1990-11-08"))
> geboren[2] - geboren[1]
Time difference of 3493 days
```

5.3.4.2.6 Elemente benennen

Beim Erzeugen eines Vektors über die Verkettungsfunktion kann man die Elemente benennen, z.B.:

```
> b <- c(one=1, too=2)
> b
one too
  1   2
```

Über die Funktion **names()** kann man die Namen der Elemente ermitteln,

```
> names(b)
[1] "one" "too"
```

und auch ändern, z.B.:

```
> names(b) <- c("alpha", "beta")
> b
alpha  beta
    1     2
```

Ein Elementname lässt sich an Stelle des zugehörigen Indexwerts verwenden, z.B.:

```
> b["alpha"]
alpha
    1
```

5.3.4.2.7 Sortieren und Ränge

Die Funktion **sort()** liefert die sortierte Variante eines Argumentvektors, z.B.:

```
> v <- c(40,55,23,11,77)
> (sv = sort(v))
[1] 11 23 40 77 87
```

Mit dem Argument **decreasing** lässt sich eine *absteigende* Sortierung veranlassen, z.B.:

```
> (sv = sort(v, decreasing=TRUE))
[1] 87 77 40 23 11
```

Die Funktion **order()** liefert einen Ergebnisvektor, der zu den Elementen des sortierten Vektors ihre Indexpositionen im Argumentvektor angibt, z.B.:

```
> (ov = order(v))
[1] 4 3 1 5 2
```

Das *i*-te Element von `ov` gibt an, welche Indexposition das Element mit dem Rang *i* (das *i*-te Element von `sv`) im Originalvektor `v` besitzt. Verwendet man das **order()**-Ergebnis als Indexvektor (vgl. Abschnitt 5.3.6.3) auf den ursprünglichen Vektor an, dann resultiert das **sort()**-Ergebnis, z.B.:

```
> v[ov]
[1] 11 23 40 55 77
```

Auch die Funktion **order()** kennt das Argument **decreasing**.

Über die Funktion **rank()** erhält man einen Ergebnisvektor mit den Rängen der Elemente des Argumentvektors, z.B.:

```
> rank(v)
[1] 3 4 2 1 5
```

Zur Veranschaulichung der drei Funktionen **sort()**, **order()** und **rank()** betrachten wir jeweils das erste Element des Ergebnisvektors (bei aufsteigender Sortierung in **sort()** und **order()**):

- `sort(v)[1]` enthält den Wert des kleinsten Elements von `v`
  ```
  > sort(v)[1]
  [1] 11
  ```
- `order(v)[1]` enthält die Indexposition des kleinsten Elements von `v`
  ```
  > order(v)[1]
  [1] 4
  ```
- `rank(v)[1]` enthält den Rangplatz des ersten Elements von `v`
  ```
  > rank(v)[1]
  [1] 3
  ```

5.3.4.2.8 Elemente replizieren

Die **rep()**-Funktion leistet eine Replikation der Elemente von Vektoren (oder auch von Faktoren und Listen, siehe unten). Ihre wichtigsten Argumente sind:

- **x**
  Objekt mit den zu wiederholenden Elementen
- **times**
  Ist das **times**-Argument ein Vektor mit Länge 1, legt es eine identische Anzahl von Wiederholungen für alle **x**-Elemente fest. Ist das **times**-Argument ein Vektor mit der Länge von **x**, legt es für jedes **x**-Element einen individuellen Wiederholungsfaktor fest.

Beispiel:

```
> v <- c(1,2,3)
> rep(v,2)
[1] 1 2 3 1 2 3
```

5.3.4.3 Faktor

Ein Faktor in **R** ist das Analogon zu einer kategorialen (nominalen oder ordinalen) Variablen in SPSS. Als Designvariable in einer Analysefunktion (z.B. **lm()**) wird ein Faktor automatisch korrekt behandelt und durch Kodiervariablen repräsentiert.

Für den Indexzugriff auf einzelne Elemente gelten bei Faktoren dieselben Regeln wie bei Vektoren (siehe Abschnitt 5.3.4.2.2).

Ein Faktor wird aus einem Vektor mit Modus **numeric** oder **character** erstellt und hat selbst stets den Modus **numeric**.

5.3.4.3.1 Nominale Faktoren

Einen **nominalen Faktor** erstellt man aus einem Vektor mit der Funktion **factor()**. Das folgende Beispiel verwendet einen Vektor mit Modus **character**:

```
> nf <- factor(c("A", "C", "B", "A", "A", "C", "B"))
> nf
[1] A C B A A C B
Levels: A B C
```

Die Funktion **levels()** liefert die verschiedenen Werte eines Faktors als Vektor mit Elementen vom Typ **character**, z.B.:

```
> levels(nf)
[1] "A" "B" "C"
```

Mit der **str()**-Funktion stellt man fest, dass die Kategorien eines Faktors intern durch natürliche Zahlen kodiert werden:

```
> str(nf)
 Factor w/ 3 levels "A","B","C": 1 3 2 1 1 3 2
```

Auch aus einem *numerischen* Vektor lässt sich ein Faktor erstellen. Im folgenden Beispiel wird das optionale Argument **labels** der Funktion **factor()** zum Etikettieren der Faktorstufen verwendet (vergleichbar mit den Wertelabels in SPSS):

```
> gruppe <- factor(c(1,2,2,1,1,3,3), labels=c("KG","EG1","EG2"))
> gruppe
[1] KG  EG1 EG1 KG  KG  EG2 EG2
Levels: KG EG1 EG2
```

Fehlt das Argument **labels**, dienen die Werte auch als Etiketten.

Wenn der Faktor in einem Modell als unabhängige Variable zum Einsatz kommt, wird die erste Kategorie als Referenz verwendet.

Über das optionale Argument **levels** der Funktion **factor()** lassen sich folgende Effekte erzielen:

- Beim Erstellen eines Faktors kann man die Reihenfolge der Faktorstufen ändern, also insbesondere die Referenzkategorie beliebig festlegen. Im folgenden Beispiel werden die Fälle mit dem Wert 3 der Kontroll- bzw. Referenzgruppe zugeordnet, z.B.:
```
> gruppe <- factor(c(1,2,2,1,1,3,3), levels=c(3,1,2), labels=c("KG","EG1", "EG2"))
> str(gruppe)
Factor w/ 3 levels "KG","EG1","EG2": 2 3 3 2 2 1 1
```
- Man kann einzelne Werte des numerischen Datenvektors ausschließen (als fehlende Werte behandeln), in dem man sie im **levels**-Argument weglässt. Im folgenden Beispiel werden Fälle mit dem Wert 3 ausgeschlossen:
```
> gruppe2 <- factor(c(1,2,2,1,1,3,3), levels=c(1,2), labels=c("KG","EG"))
> str(gruppe2)
Factor w/ 2 levels "KG","EG": 1 2 2 1 1 NA NA
```

Die Faktorstufen müssen über ihr Etikett angesprochen werden, z.B.

richtig: `syms[gruppe=="KG"] <- 5`
falsch: `syms[gruppe==1] <- 5`

5.3.4.3.2 Ordinale Faktoren

Einen **ordinalen Faktor** erstellt man mit der Funktion **ordered()**, z.B.:

```
> of <- ordered(c("A", "C", "B", "A", "A", "C", "B"))
> of
[1] A C B A A C B
Levels: A < B < C
```

5.3.4.4 Matrix

Eine Matrix ist zur Aufnahme empirischer Daten geeignet, wenn alle Werte vom selben Datentyp sind (z.B. **numeric**). Im Vergleich zur später vorzustellenden Datentabelle (siehe Abschnitt 5.3.4.7) ist die Matrix ...

- weniger flexibel,
- aber Speicherplatz schonender.

Damit eignet sich die Matrix speziell für Anwendungen aus dem Bereich der linearen Algebra, wobei viele Matrixfunktionen zur Verfügung stehen (siehe Abschnitt 10.2).

Sollen numerische Variablen in einer Datentabelle durch eine **R**-Funktion verarbeitet werden, die nur Matrizen unterstützt, dann hilft die Funktion **as.matrix()** weiter.

5.3.4.4.1 Matrix aus einem Vektor erstellen

Wird für die Elemente eines Vektors eine Doppelindizierung definiert, resultiert eine Matrix. In der Regel verwendet man zum Erzeugen die Funktion **matrix()** mit den folgenden Argumenten:

- **data**
  Hier ist ein Vektor anzugeben.
- **nrow**
  Anzahl der Zeilen

- **ncol**
  Anzahl der Spalten
- **byrow**
  Zeilen- statt Spaltendominanz beim Verteilen der Vektorelemente auf die Matrixzellen (siehe unten)

Man muss nur die Zeilen- *oder* die Spaltenzahl angeben. Um auf die Spaltenangabe zu verzichten, lässt man den Parameter **ncol** weg, z.B.:

```
> matz <- matrix(c(1,2,3,4,5,6), 2)
> matz
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Soll nur die Spaltenzahl genannt werden, ist **ncol** als **Namensargument** (statt als Positionsargument) zu verwenden, z.B.:

```
> matz <- matrix(c(1,2,3,4,5,6), ncol=2)
> matz
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Wie die Beispiele zeigen, verwendet **R** per Voreinstellung ein Indizierungsschema mit **Spaltendominanz**, d.h. die verfügbaren Vektorelemente füllen zunächst die Spalte 1 von oben nach unten, dann die Spalte 2 usw. Sollen statt dessen die Zeilen nacheinander befüllt werden, ist für das Argument **byrow** der Wert **TRUE** anzugeben, z.B.:

```
> matz <- matrix(c(1,2,3,4,5,6), 2, byrow=TRUE)
> matz
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

5.3.4.4.2 Vektoren und/oder Matrizen zu einer neuen Matrix koppeln

Mit den Funktionen **rbind()** und **cbind()** lassen sich dimensional passende Matrizen und/oder Vektoren hintereinander bzw. nebeneinander koppeln, z.B.:

```
> matz2 <- matz + 3
> matz2
     [,1] [,2] [,3]
[1,]    4    5    6
[2,]    7    8    9
> Fertig <- rbind(cbind(matz,matz2),c(1,1,1,1,1,1))
> Fertig
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    4    5    6    7    8    9
[3,]    1    1    1    1    1    1
```

Werden *benannte* Vektoren gekoppelt, resultiert eine Matrix mit benannten Spalten bzw. Zeilen, z.B.:

```
> v1 <- c(1,2,3)
> v2 <- c(4,5,6)
> m <- cbind(v1,v2)
> m
```

```
     v1 v2
[1,]  1  4
[2,]  2  5
[3,]  3  6
```

5.3.4.4.3 Eigenschaften einer Matrix

Über die Struktur einer Matrix informiert die Funktion **str()**, z.B.:

```
> str(matz)
 num [1:2, 1:3] 1 4 2 5 3 6
```

Wir erfahren im Beispiel, dass die Matrix 2 Zeilen und 3 Spalten hat. Ihre Elemente werden spaltendominant aufgelistet.

Die Funktion **dim()** liefert die Maximalindizes zu den beiden Dimensionen, z.B.

```
> dim(matz)
[1] 2 3
```

Über die Funktionen **nrow()** bzw. **ncol()** kann man die Anzahl der Zeilen bzw. Spalten auch separat ermitteln, z.B.:

```
> nrow(matz)
[1] 2
> ncol(matz)
[1] 3
```

Über die Funktionen **rownames()** bzw. **colnames()** lassen sich die Zeilen- bzw. Spaltennamen einer Matrix setzen oder ermitteln, z.B.:

```
> rownames(matz) <- c('A', 'B', 'C')
> colnames(matz) <- c('I', 'II')
> matz
  I II
A 1  4
B 2  5
C 3  6
> rownames(matz)
[1] "A" "B" "C"
```

Wie bei einem Vektor gilt bei einer Matrix für die den Datentyp der Elemente:

- Alle Elemente müssen vom selben Datentyp sein.
- Neben **numeric** sind auch alternative Datentypen möglich (z.B. **character**, **logical**).

5.3.4.4.4 Matrizen transponieren

Zum **Transponieren** einer Matrix steht die Funktion **t()** bereit, z.B.:

```
> matz <- matrix(c(1,2,3,4,5,6), 2, byrow=TRUE)
> matz
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
> tmatz <- t(matz)
> tmatz
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> str(tmatz)
 num [1:3, 1:2] 1 2 3 4 5 6
```

5.3.4.4.5 Funktionen auf Zeilen oder Spalten anwenden

Häufig soll eine Funktion (z.B. **max()** zur Bestimmung des Maximums) auf alle Zeilen oder Spalten einer Matrix angewendet werden. In dieser Situation bietet die Funktion **apply()** eine bequeme und effiziente Lösung. Ihre Argumente sind:

- **X**
  Hier ist die zu analysierende Matrix anzugeben.
- **MARGIN**
  Hier wählt man zwischen den Werten
  - 1 für Zeilenbezug und
  - 2 für Spaltenbezug
- **FUN**
  Hier ist der Name einer Funktion ohne Parameterliste anzugeben, die Vektoren verarbeiten kann.

Im folgenden Beispiel werden die maximalen Spaltenwerte ermittelt:

```
> daten <- matrix(c(4,5,7,4,3,7,8,9,  3,4,6,2,3,7,3,4),ncol=2)
> daten
     [,1] [,2]
[1,]    4    3
[2,]    5    4
[3,]    7    6
[4,]    4    2
[5,]    3    3
[6,]    7    7
[7,]    8    3
[8,]    9    4
> apply(daten, 2, max)
[1] 9 7
```

5.3.4.5 Array

Die Klasse Array erweitert die Klasse Matrix, indem statt zwei beliebig viele Dimensionen erlaubt sind. Wie bei der Matrix müssen auch beim Array alle Elemente denselben elementaren Datentyp besitzen (z.B. **numeric**, **logical**, **character**).

Man erstellt einen Array über die Funktion **array()** mit den folgenden Parametern:

- **data**
  Vektor mit den Elementen
- **dim**
  Hier ist ein Vektor anzugeben, der durch seine Länge die Anzahl der Array-Dimensionen und durch seine Elemente den Maximalindex pro Dimension angibt.

Beispiel:

```
> Arri <- array(c(1,2,3,4,5,6,7,8), c(2,2,2))
> Arri
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

5.3.4.6 Liste

Eine Liste besteht wie ein Vektor aus einer Anzahl von Elementen, die aber *nicht* vom selben Typ sein müssen. Für die Elemente sind alle Typen erlaubt, insbesondere auch der Typ Liste. Weil die Liste somit eine enorme Flexibilität bietet, wird sie von vielen **R**-Funktionen zur Organisation ihrer Ausgaben verwendet. Die von der Funktion **length()** gelieferte Länge einer Liste wird durch die Anzahl ihrer Elemente auf oberster Ebene festgelegt. Um die strukturelle Flexibilität bei den Bestandteilen einer Liste zu betonen, spricht man von den *Komponenten* einer Liste (Muenchen 2011, S. 83).

5.3.4.6.1 Liste erstellen

Man erstellt eine Liste über die Funktion **list()**, z.B.:

```
> lis <- list(c(1,2,3), matrix(c(1,2,3,4), 2), list("Jetzt schlägt es", 13))
> lis
[[1]]
[1] 1 2 3

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[[3]][[1]]
[1] "Jetzt schlägt es"

[[3]][[2]]
[1] 13
```

Die Liste `Lis` enthält die folgenden drei Komponenten

- einen Vektor
- eine Matrix
- eine Liste

und hat damit die Länge 3:

```
> length(lis)
[1] 3
```

Beim Erzeugen einer Liste kann man ihre Elemente benennen, z.B.:

```
> lis <- list(vektor=c(1,2,3), matrix=matrix(c(1,2,3,4), 2), liste=list("Jetzt schlägt es", 13))
```

Wie bei einem Vektor kann man auch bei einer Liste über die Funktion **names()** die Komponenten benennen bzw. vorhandene Labels ermitteln, z.B.:

```
> names(lis) <- c("vektor", "matrix", "liste")
```

Sind Namen vorhanden, erscheinen diese in der Ausgabe anstelle von Indexnummern, z.B.:

```
> lis
$vektor
[1] 1 2 3

$matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$liste
$liste[[1]]
[1] "Jetzt schlägt es"

$liste[[2]]
[1] 13
```

5.3.4.6.2 Zugriff auf Bestandteile einer Liste

Für den Zugriff auf die Komponenten einer Liste ist der **[[ ]]** - Operator zu verwenden, z.B.:

```
> lis[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Sind Namen vorhanden, können diese für den Zugriff auf die Komponenten verwendet werden, entweder über den **[[ ]]** – Operator

```
> lis[["matrix"]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

oder über den **$**-Operator:

```
> lis$matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Man muss schon genau hinsehen, um den Wirkungsunterschied zwischen dem Komponentenauswahl-Operator **[[ ]]** und dem allgemeinen Indizierungs-Operator **[ ]** bei einer Liste zu erkennen:

```
> lis[[1]]
[1] 1 2 3
```

```
> lis[1]
$Vektor
[1] 1 2 3
```

`lis[[1]]` ist die erste Komponente in der Liste (ein Vektor mit numerischen Elementen). `lis[1]` ist hingegen eine Liste, die allerdings nur noch die erste Komponente von `Lis` enthält. Mit der **class()**-Funktion befragt, nennen die beiden Objekte ihren Typ:

```
> class(lis[[1]])
[1] "numeric"
> class(lis[1])
[1] "list"
```

Das Verhalten des allgemeinen Indizierungsoperators **[ ]** in **R** ist absolut konsistent. Angewandt auf einen Vektor liefert er einen neuen Vektor mit Länge 1 und angewandt auf eine Liste liefert er eine neue Liste der Länge 1. Verwendet man einen Indexvektor (siehe Abschnitt 5.3.6.3) in Kombination mit dem allgemeinen Indizierungs-Operator lässt sich ein Teilvektor oder eine Teilliste mit einer Länge größer 1 gewinnen.

Um ein bestimmtes Element aus einer Listenkomponente anzusprechen, lässt man die beiden Indexoperatoren aufeinander folgen, z.B.:

```
> lis[[1]][3]
[1] 3
```

Es ist aber auch folgende Schreibweise erlaubt:

```
> lis[[c(1,3)]]
[1] 3
```

5.3.4.7 Datentabelle

Ein Data Frame (in Anlehnung an Ligges 2007 übersetzt mit: *Datentabelle*) in **R** entspricht einem Datenblatt (mit einer (Fälle × Variablen)-Datenmatrix) in SPSS. Es ist eine *Liste* von Vektoren, Faktoren und Matrizen, die *alle gleich lang* sind.

Fast alle (in einer SPSS-Datendatei oder einer Textdatei) zum Import in **R** bereitstehende Datensätze sind als Tabelle nach dem (Fälle × Variablen) - Prinzip aufgebaut, d.h.:

- In einer Zeile stehen alle von einem Fall stammenden Daten.
- In einer Spalte steht eine Variable, die für jeden Fall einen Wert (oder einen Indikator für fehlende Werte) enthält.

In einem größeren Forschungsprojekt werden eventuell *mehrere* Datentabellen benötigt.

5.3.4.7.1 Datentabelle erzeugen

Man kann eine Datentabelle mit der Funktion **data.frame()** erstellen, z.B.:

```
> alter <- c(45, 32, 58)
> geschlecht <- factor(c(1, 1, 2))
> dt <- data.frame(alter, geschlecht)
```

```
> dt
  alter geschlecht
1    45          1
2    32          1
3    58          2
```

Ein Vektor mit Elementen vom Typ **character** wird bei Aufnahme in eine Datentabelle per Voreinstellung in einen Faktor umgewandelt, z.B.:

```
> augenfarbe <- c("blau","braun","grün")
> dt <- data.frame(alter, geschlecht, augenfarbe)
> str(dt)
'data.frame':   3 obs. of  3 variables:
 $ alter     : num  45 32 58
 $ geschlecht: Factor w/ 2 levels "1","2": 1 1 2
 $ augenfarbe: Factor w/ 3 levels "blau","braun",..: 1 2 3
```

Ist diese Wandlung unerwünscht, kann sie folgendermaßen verhindert werden:

- für einen einzelnen **character**-Vektor durch Schachteln in einen Aufruf der Funktion **I()**, z.B.:

```
> values <- c(1, 2, 3)
> names <- c("A", "B", "C")
> tcv <- data.frame(values, I(names))
```

- für den gesamten **data.frame()** - Aufruf durch den Wert **FALSE** für das Argument **stringsAsFactors**, z.B.:

```
> tcv <- data.frame(values, names, stringsAsFactors=FALSE)
```

Mit dem Typ der in eine Datentabelle aufgenommenen Vektoren legt man das **Messniveau** der repräsentierten Merkmale fest:

- Für quantitative (kontinuierliche, intervallskalierte) Merkmale verwendet man numerische Vektoren.
- Für rangskalierte Merkmale verwendet man ordinale Faktoren (vgl. Abschnitt 5.3.4.3.2).
- Für nominalskalierte Merkmale verwendet man nominale Faktoren (vgl. Abschnitt 5.3.4.3.1).
- Ein Zeichenkettenvektor (z.B. mit dem Namen der Fälle) wird von **R** als nominalskaliert behandelt.

5.3.4.7.2 Eigenschaften einer Datentabelle ermitteln und ändern

Über die aktuelle Struktur einer Datentabelle informiert die Funktion **str()**:

```
> str(dt)
'data.frame':   3 obs. of  2 variables:
 $ alter     : num  45 32 58
 $ geschlecht: Factor w/ 2 levels "1","2": 1 1 2
```

Wie bei einer Matrix kann man über die Funktionen **dim()**, **nrow()** bzw. **ncol()** die Anzahl der Fälle bzw. Variablen ermitteln, z.B.:

```
> dim(dt)
[1] 3 2

> nrow(dt)
[1] 3

> ncol(dt)
[1] 2
```

Die Namen der Spalten (Variablen) können über die Funktion **names()** abgefragt und geändert werden, z.B.:

```
> names(dt)
[1] "alter"      "geschlecht"
> names(dt) <- c("age", "gender")
```

Die Namen der Zeilen (Fälle) können über die Funktion **row.names()** abgefragt und geändert werden, z.B.:

```
> row.names(dt)
[1] "1" "2" "3"
> row.names(dt) <-("alpha", "beta", "gamma")
```

Per Voreinstellung werden laufende Nummern verwendet.

5.3.4.7.3 Bestandteile einer Datentabelle ansprechen

Auf eine einzelne Variable greift man über den $-Operator zu, z.B.:

```
> mean(dt$alter)
[1] 45
```

Befindet sich eine Matrix als Komponente in einer Datentabelle, werden ihre Spalten wie Vektoren behandelt, anzusprechen über den Matrixnamen mit angehängter Spaltennummer, z.B.:

```
> dfm <- data.frame(mat = matrix(c(1,2,3,4,5,6),ncol=2), vec = factor(c(1,1,2)))
> dfm
  mat.1 mat.2 vec
1     1     4   1
2     2     5   1
3     3     6   2
> dfm$mat.2
[1] 4 5 6
```

Zur Ansprache einzelner Werte kann bei Datentabellen (abweichend von gewöhnlichen Listen) die Notation der Matrizen verwendet werden, z.B.:

```
> dt <- data.frame(alter = c(45, 32, 58), geschlecht = factor(c(1, 1, 2)))
> dt[3,1]
[1] 58
```

Weitere Möglichkeiten zur Auswahl von Bestandteilen einer Datentabelle werden in Abschnitt 5.3.6 vorgestellt.

5.3.4.7.4 Variablen ergänzen oder entfernen

Um eine zusätzliche Variable in eine Datentabelle

```
> dt <- data.frame(alter=c(45, 32, 58), geschlecht=factor(c(1, 1, 2)))
```

aufzunehmen, kann man die Funktion **data.frame()** erneut aufrufen. Man verwendet die vorhandene Datentabelle als erstes Argument, ergänzt die neue Variable als zweites Argument und weist der Datentabelle die Funktionsrückgabe zu, z.B.:

```
> dt <- data.frame(dt, bildung = factor(c(3,4,2)))
```

Eine alternative Möglichkeit besteht darin, den Namen der neuen Variablen per $-Syntax an den Datentabellennamen anhängen und per Zuweisungsoperator die Werte folgen zu lassen, z.B.:

```
> dt$bildung <- factor(c(3,4,2))
> dt
  alter geschlecht bildung
1    45          1       3
2    32          1       4
3    58          2       2
```

Eine einfache Möglichkeit, eine Variable aus einer Datentabelle zu entfernen, besteht darin, dem Variablennamen den Wert **NULL** zuzuweisen, z.B.:

```
> dt$bildung <- NULL
> dt
  alter geschlecht
1    45          1
2    32          1
3    58          2
```

5.3.4.7.5 Datentabelle in den Suchpfad der R-Sitzung aufnehmen

Um die Angabe des Datentabellennamens einzusparen, kann man einen Data Frame per **attach()** in den Suchpfad der **R**-Sitzung aufnehmen, z.B.:

```
> attach(dt)
> mean(alter)
[1] 45
```

Ein Aufruf der Funktion **search()** zeigt, dass sich die Datentabelle nun an 2. Position im Suchpfad befindet, direkt hinter der globalen Umgebung der **R**-Sitzung (mit den Workspace-Objekten):

```
> search()
 [1] ".GlobalEnv"        "dt"                "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Bei Namensgleichheit gewinnt also das Objekt in `dt`, sofern sich der Konkurrent nicht in der globalen Umgebung befindet.

Mit der Funktion **detach()** lässt sich eine Datentabelle wieder aus dem Suchpfad entfernen, z.B.:

```
> detach(dt)
```

Es ist zu beachten, dass **attach()** *Kopien* der Variablen erzeugt, so dass sich Schreibzugriffe *nicht* auf die Originale auswirken (Wollschläger 2010, S. 126), z.B.:

```
> dt <- data.frame(alter=c(45, 32, 58), geschlecht=factor(c(1, 1, 2)))
> attach(dt)
> alter
[1] 45 32 58
> alter[1] <- 99
> alter
[1] 99 32 58
> dt$alter
[1] 45 32 58
```

Die Kopien persistieren auch nach der **detach()** - Anweisung, z.B.:

```
> detach(dt)
> alter
[1] 99 32 58
```

Wegen des großen Fehlerrisikos warnen viele **R**-Kenner vor der **attach()** - Funktion (z.B. Bliese 2013, S. 11; Muenchen 2011, S. 422ff).

Wenn in *einem* Funktionsaufruf mehrere Variablen aus einer Datentabelle angesprochen werden müssen, kann die **with()** - Funktion eine Vereinfachung bewirken. Im folgenden Aufruf der **hist()**-Funktion zur Erstellung eines Histogramms für eine Teilstichprobe (vgl. Abschnitt 9.2.4.5) stammen die Variablen `x` und `treat` aus der Datentabelle `dtlongname`:

```
> with(dtlongname,hist(x[treat==0]))
```

Im Vergleich zur alternativen Schreibweise

```
> hist(dtlongname$x[dtlongname$treat==0])
```

ist allerdings nur ein kleiner Einspareffekt festzustellen. Der **with()**-Rahmen muss für jeden Funktionsaufruf wiederholt werden und trägt durch das zusätzliche Klammernpaar nicht zur Übersichtlichkeit bei. Daher wird die syntaktisch weit bequemere **attach()** - Funktion trotz der damit verbundenen Risiken vielfach doch verwendet.

Bei Modellierungsfunktionen kann man im **data**-Argument eine Datentabelle angeben und deren Variablen in anderen Argumenten mit einfachen Namen ansprechen, z.B.:

```
> lm(formula = y ~ x, data = casedata)
```

5.3.4.7.6 Funktionen auf Variablen oder Fälle anwenden

Wie bei Matrix-Objekten (siehe Abschnitt 5.3.4.4.5) lässt sich auch bei Datentabellen per **apply()** eine Funktion auf alle Spalten oder alle Zeilen anwenden. Im folgenden Beispiel werden die maximalen Werte der Variablen in einer Datentabelle ermittelt:

```
> dt <- data.frame(alter = c(45,32,58), groesse=c(166,167,178))
> apply(dt, 2, max)
  alter groesse
     58     178
```

Mit dem zweiten Parameter legt man fest, ob die Funktion auf die Zeilen (Wert = 1) oder auf die Spalten (Wert = 2) wirken soll.

5.3.5 Fehlende Werte

Bei beliebigen Datentypen dient **NA** (*Not Available*) als Ersatz für fehlende Werte, z.B. bei einer Datentabelle mit zwei numerischen Vektoren

```
> dt$alter[1] <- NA
> dt
  alter geschlecht
1    NA          1
2    32          1
3    58          2
```

oder bei einem Vektor mit Modus **character**

```
> like <- c("y", NA, "y", "n")
> like
[1] "y" NA  "y" "n"
```

Es ist zu beachten, dass **NA** auch bei Variablen mit **character**-Modus *ohne* Anführungszeichen geschrieben wird.

Über die Funktion **is.na()** lässt sich überprüfen, ob der Ersatzwert **NA** vorliegt, z.B.:

```
> is.na(dt$alter[1])
[1] TRUE
```

Bei einem mehrelementigen Objekt (z.B. Vektor, Matrix, Datentabelle) als Argument erhält man ein strukturgleiches Ergebnisobjekt, z.B.:

```
> is.na(dt)
     alter geschlecht
[1,]  TRUE      FALSE
[2,] FALSE      FALSE
[3,] FALSE      FALSE
```

Über die Funktion **any()** kann man feststellen, ob überhaupt Werte fehlen, z.B.:

```
> any(is.na(dt))
[1] TRUE
```

Die Anzahl fehlender Werte lässt sich mit der Funktion **sum()** ermitteln, weil die logischen Werte **TRUE** und **FALSE** intern als 1 und 0 gespeichert werden, z.B.:

```
> sum(is.na(dt))
[1] 1
```

Misslingt eine Berechnung (z.B. 0/0), resultiert der Ersatzwert **NaN** (*Not a Number*), z.B.

```
> x<- 0/0
> x
[1] NaN
```

Über die Funktion **is.nan()** lässt sich überprüfen, ob der Ersatzwert **NaN** vorliegt, z.B.:

```
> is.nan(x)
[1] TRUE
```

In der Regel benimmt sich ein **NaN** wie **NA**, und entsprechend liefert die Funktion **is.na()** auch beim Ersatzwert **NaN** das Ergebnis **TRUE**:

```
> is.na(x)
[1] TRUE
```

Gelegentlich ist es sinnvoll, sich auf die Fälle mit einem vollständigen Datensatz zu beschränken. Von der Funktion **na.omit()** erhält man die um Fälle mit fehlenden Werten (**NA** oder **NAN**) bereinigte Variante einer Datentabelle, z.B.:

```
> dt <- dt <- data.frame(alter = c(45,32,NA), groesse=c(NaN,167,178))
> dt <- na.omit(dt)
> dt
  alter groesse
2    32     167
```

Enthält ein Vektor einen fehlenden Wert (**NA** oder **NaN**) liefern statistische Funktionen wie **sum()** oder **mean()** das Ergebnis **NA** oder **NaN**, z.B.:

```
> a <- c(1, NA, 3)
> mean(a)
[1] NA
```

Soll stattdessen aus den vorhandenen Argumenten ein Ergebnis ermittelt werden, ist das Argument **na.rm** auf den Wert **TRUE** zu setzen, z.B.:

```
> mean(a, na.rm=TRUE)
[1] 2
```

5.3.6 Indexzugriff

5.3.6.1 Zugriff auf einzelne Elemente

Zugriff auf **einzelne Elemente** (Indexstart mit 1):

- Bei einem Vektor: [*i*]
  Beispiel:
  ```
  > i <- 3
  > v <- c(1,2,3)
  > v[i]
  [1] 3
  ```
- Bei einer Matrix: [*i*, *j*]
  Beispiel:
  ```
  > m <- matrix(c(1,2,3,4,5,6),3)
  > m
       [,1] [,2]
  [1,]    1    4
  [2,]    2    5
  [3,]    3    6
  > m[2,2]
  [1] 5
  ```
- Bei einer Liste: [[*i*]],
  Bei einer Liste mit benannten Elementen: [["*name*"]] oder *liste*$*name*
  Beispiel:
  ```
  > lst <- list(name="Brgl", vorname="Thea", anzKinder=4, alterKinder=c(12,10,8,2))
  ```
  Ansprache per Index
  ```
  > lst[[2]]
   [1] "Thea"
  ```
  Ansprache per Elementname:
  ```
  > lst[["Vorname"]]
  [1] "Thea"
  > lst$alterKinder[3]
  [1] 8
  ```

5.3.6.2 Zugriff auf einen Zeilen- oder Spaltenvektor aus einer Matrix oder Datentabelle

Um die *k*-te Zeile oder Spalte einer Matrix auszuwählen, verwendet man den Indexausdruck **[*k*,]** oder **[, *k*]**, z.B.

```
> m <- matrix(c(1,2,3,4,5,6), 2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m[2,]
[1] 2 4 6
> m[,1]
[1] 1 2
```

Analog kann man einen Fall bzw. eine Variable aus einer Datentabelle wählen, z.B.:

```
> daten <- data.frame(alter=c(45,55,NA,58), geschlecht=factor(c(1,2,1,2)))
> daten[1,]
  alter geschlecht
1    45          1
```

Die von Datentabellen bekannte $-Notation zur Auswahl von Spalten (Variablen)

```
> Daten$alter
[1] 45 55 NA 58
```

klappt bei Matrizen mit benannten Spalten nicht, z.B.:

```
> v1 <- c(1,2,3)
> v2 <- c(4,5,6)
> m <- cbind(v1,v2)
> m
     v1 v2
[1,]  1  4
[2,]  2  5
[3,]  3  6
> m$v1 # Fehler!
Fehler in m$v1 : $ operator is invalid for atomic vectors
```

Auf folgende Weise lassen sich vorhandene Spaltenlabel bei Matrizen aber doch zur Spaltenauswahl nutzen:

```
> m[,"v1"]
[1] 1 2 3
```

5.3.6.3 Indexvektoren

Über ein Vektor-Argument für den allgemeinen Index-Operator **[ ]** wählt man eine Teilmenge der Elemente ...

- in einem Vektor
- in den Zeilen oder Spalten einer Matrix
- in einer Datentabelle
- in einer Liste

Als Indexvektor sind u.a. erlaubt:

- Numerischer Indexvektor
  Durch einen Vektor mit ausschließlich positiven Einträgen werden die Elemente mit den entsprechenden Indexwerten ausgewählt, z.B.:
  ```
  > v <- c(11,7,13,54,76)
  > v[c(1,3,4)]
  [1] 11 13 54
  ```
  Im nächsten Beispiel wird per Sequenzoperator ein Indexvektor für die Auswahl von Matrixspalten gebildet:
  ```
  > m[,1:2]
       [,1] [,2]
  [1,]    1    3
  [2,]    2    4
  ```
  Durch einen Vektor mit ausschließlich *negativen* Einträgen werden die Elemente mit den entsprechenden Indexwerten ausgeschlossen, z.B.:

```
> v[c(-1,-3,-4)]
[1]  7 76
```

- Logischer Indexvektor
  Durch einen Indexvektor mit dem Modus **logical** werden alle Indexelemente mit den Wert **TRUE** ausgewählt, z.B.:
  ```
  > t <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
  > v[t]
  [1] 11  7
  ```
  Im folgenden Beispiel werden mit Hilfe des Modulo-Operators (vgl. Abschnitt 5.3.7.1) die Elemente mit geradzahligem Wert gewählt:
  ```
  > v[v %% 2 == 0]
  [1] 54 76
  ```

Über eine Wertzuweisung an den Indexvektorausdruck lassen sich die ausgewählten Elemente ändern. Im folgenden Beispiel werden die negativen Werte eines Vektors auf 0 gesetzt:

```
> v <- c(-1,3,5,-7,2)
> v[v<0] <- 0
> v
[1] 0 3 5 0 2
```

Enthält ein logischer Indexvektor **NA**-Werte, sollte er durch die Funktion **which()** (siehe Abschnitt 5.3.7.3) in einen numerischen Indexvektor überführt werden (Wollschläger, 2010, S. 38). **which()** liefert einen Vektor mit den Indexpositionen der **TRUE**-Werte im Argumentvektor, z.B.:

```
> daten <- data.frame(alter=c(45,55,NA,58), geschlecht=factor(c(1,2,1,2)))
> (indLog <- Daten$alter > 50)
[1] FALSE  TRUE    NA  TRUE
> (indNum <- which(indLog))
[1] 2 4
```

Über den logischen Indexvektor gelingt es im Beispiel nicht, eine Datentabelle mit den Positivfällen zu extrahieren:

```
> Daten[indLog,]
   alter geschlecht
2     55          2
NA    NA       <NA>
4     58          2
```

Mit der **which()**-Rückgabe wird dieses Ziel hingegen erreicht:

```
> Daten[indNum,]
  alter geschlecht
2    55          2
4    58          2
```

Weitere Details zu Indexvektoren erläutert das **R** Development Core Team (2014, Abschnitt 3.4.1, S. 16).

5.3.6.4 Indexmatrizen

Einen Vektor mit einer Auswahl der Elemente einer Matrix gewinnt man mit einer Indexmatrix, die 2 Spalten aufweist, so dass jede Zeile die Indexpositionen eines ausgewählten Elements der Ausgangsmatrix enthält, z.B.:

```
> m <- matrix(c(1,2,3,4), 2)
> m
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> im <- matrix(c(1,1,2,2), ncol=2, byrow=TRUE)
> im
     [,1] [,2]
[1,]    1    1
[2,]    2    2
> m[im]
[1] 1 4
```

5.3.7 Operatoren

In **R** arbeiten die Operatoren meist **elementweise**.

5.3.7.1 Arithmetische Operatoren

Symbole für die arithmetischen Operationen:

+	Addition
-	Subtraktion
*****	Multiplikation
/	Division
^ oder **	Potenzieren
%%	Modulo (Divisionsrest)

Diese Operatoren werden auf Vektoren und Matrizen *elementweise* angewendet, wie das folgende Beispiel zeigt:

```
> M <- matrix(c(1,2,3,4), 2)
> M
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> IM <- solve(M)
> IM
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> M*IM
     [,1] [,2]
[1,]   -2  4.5
[2,]    2 -2.0
```

Obiger Aufruf der Funktion **solve()** liefert die inverse Matrix `IM` zum Argument `M` (vgl. Abschnitt 10.2). Weil der Produktoperator ***** elementweise arbeitet, ergibt das Produkt `M*IM` *nicht* die Einheitsmatrix. Dazu muss mit dem **%*%** - Operator das *Matrixprodukt* berechnet werden:

```
> M %*% IM
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

5.3.7.2 Vergleichsoperatoren

Bei den Symbolen für die Vergleichsoperatoren in **R** ist vor allem zu beachten, dass der Identitätsoperator (wie in den Programmiersprachen C und Java) durch zwei = - Zeichen ausgedrückt wird:

==	gleich
!=	verschieden
<=	kleiner oder gleich
<	kleiner
>	größer
>=	größer oder gleich

Beispiel:

```
> a <- c(1,2,3)
> b <- c(1,4,5)
> a<b
[1] FALSE TRUE TRUE
```

Beim Identitätstest für Gleitkommazahlen müssen technisch bedingte Abweichungen von der reinen Mathematik berücksichtigt werden, z.B.:

```
> 0.1 == (10.0 - 9.9)
[1] FALSE
```

Hier hilft die Funktion **all.equal()**:

```
> all.equal(0.1, (10.0 - 9.9))
[1] TRUE
```

5.3.7.3 Logische Operatoren

Die folgenden logischen Operatoren wirken elementweise:

!	nicht
&	und
|	oder

Beispiel:

```
> a <- c(1, 3, 5)
> a > 1 & a < 5
[1] FALSE  TRUE FALSE
```

Bei elementweisen logischen Operationen (mit einem Vektor von Wahrheitswerten als Ergebnis) kann man ...

- über die Funktion **all()** prüfen, ob *alle* Ergebniselemente gleich **TRUE** sind, z.B.:
  ```
  > all(a > 1 & a < 5)
  [1] FALSE
  ```
- über die Funktion **any()** prüfen, ob mindestens ein Ergebniselement gleich **TRUE** ist, z.B.:
  ```
  > any(a > 1 & a < 5)
  [1] TRUE
  ```

Ist man an den Indexpositionen der **TRUE**-Werte interessiert, hilft die Funktion **which()** weiter, z.B.:

```
> which(a > 1 & a < 5)
[1] 2
```

Im Unterschied zu den elementweisen logischen Operatoren berücksichtigen die folgenden Operatoren nur das jeweils *erste* Element:

&&	und
||	oder

Beispiele:

```
> t1 <- c(FALSE, FALSE, TRUE)
> t2 <- c(FALSE, TRUE, FALSE)
> t1 || t2
[1]  FALSE
```

5.3.7.4 Sequenzoperator

Das Zeichen **:** steht in **R** für den Sequenzoperator, der einen Vektor aus Zahlen produziert, beginnend mit dem linken Operanden und dann im Einserabstand wachsend bis zur letzten Zahl, die den rechten Operanden nicht übertrifft, z.B.:

```
> 1:5
[1] 1 2 3 4 5
```

Mit der **seq()**-Funktion lässt sich eine alternative Schrittweite einstellen (auch eine negative), z.B.:

```
> seq(0.3, 0.36, by=0.01)
[1] 0.30 0.31 0.32 0.33 0.34 0.35 0.36
```

5.3.7.5 Recycling-Regel

Sind zwei Objekte elementweise zu verarbeiten, werden bei ungleicher Länge Elemente der kürzeren Struktur wiederverwendet, bis Längengleichheit besteht, z.B.:

```
> a <- 5
> b <- c(1,2,3)
> a+b
[1] 6 7 8
```

Ist die Länge des größeren Objekts *kein* Vielfaches der kleineren Länge, vermutet **R** ein Problem und warnt, z.B.:

```
> a <- c(5,6)
> b <- c(1,2,3)
> a+b
[1] 6 8 8
Warnmeldung:
In a + b : Länge des längeren Objektes
         ist kein Vielfaches der Länge des kürzeren Objektes
```

5.3.7.6 Zuweisungsoperatoren

Wie Sie mittlerweile aus zahlreichen Beispielen wissen, wird in **R** bevorzugt das Zeichenduo **<-** bei Wertzuweisungen verwendet.[1] In fast allen Situationen ist das = - Zeichen äquivalent.[2]

---

[1] Siehe z.B.: http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

[2] Exotische Ausnahme: Wer bei einem Funktionsaufruf in einem Aktualparameter-Ausdruck eine Variable definieren möchte, muss den offiziellen Zuweisungsoperator verwenden, z.B.:

```
> mean(g=1:5)
Fehler .....
> mean(g<-1:5)
[1] 3
> g
```

Beim empfohlenen Zweizeichen-Zuweisungsoperator kann es durch ein versehentlich eingefügtes Leerzeichen zu einem Fehler kommen, z.B.:

```
> oh < - 3
[1] TRUE
> oh
[1] -4
```

Es ist erlaubt, aber nicht üblich, den rechtsorientierten Zuweisungsoperator **->** zu verwenden, z.B.:

```
> 13 -> k
```

5.3.8 Anweisungen

Eine komplette **R**-Anweisung kann durch ein Semikolon oder eine Zeilentrennung abgeschlossen werden. Über das Semikolon ist es möglich, mehrere Anweisungen in einer Zeile unterzubringen, z.B.:

```
> g<-1:5; mean(g)
[1] 3
```

Bei einer unfertigen Anweisung erwartet **R** nach einem Zeilenwechsel die Fortsetzung der Anweisung. In dieser Situation präsentiert die graphische **R**-Bedienoberfläche (RGui) einen Fortsetzungs-Prompt, z.B.:

```
> g <-
+
```

5.3.8.1 if - Anweisungen

Durch die **if**-Anweisung kann man die Ausführung einer Anweisung von einer Bedingung abhängig machen. In der folgenden Syntaxbeschreibung sind die kursiv gesetzten Platzhalter durch zulässige Konkretisierungen zu ersetzen:

if (*Logischer Ausdruck***)**
    Anweisung

Die Anweisung wird nur dann ausgeführt, wenn der logische Ausdruck den Wert **TRUE** besitzt.

Beispiel:

```
> if(a >= 1) b <- log(a)
```

5.3.8.2 if-else - Anweisung

Soll auch dann etwas passieren, wenn der steuernde logische Ausdruck den Wert **FALSE** besitzt, erweitert man die **if**-Anweisung um eine **else**-Klausel:

if (*Logischer Ausdruck***)**
    Anweisung 1
else
    Anweisung 2

Beispiel:

```
> if(a >= 1) b <- log(a) else b <- 0
```

```
[1] 1 2 3 4 5
```

5.3.8.3 Wiederholungsanweisungen

Durch eine **for**-Schleife sorgt man dafür, dass eine Anweisung wiederholt ausgeführt wird, wobei in der Anweisung eine Schleifenvariable *Var* auftritt, die beim *i*-ten Schleifendurchgang das *i*-te Element eines Objekts als Wert annimmt:

for (*Var* **in** *Objekt***)**
    Anweisung

Beispiel:

```
> sq <- 0; a <- c(4,7,8)
> for (i in a) sq <- sq + i^2
> sq
[1] 129
```

In **R** kann und sollte man Schleifen weitgehend vermeiden, um eine elegante und performante Programmierung zu erhalten. Auch die **for**-Schleife im letzten Beispiel lässt sich leicht ersetzen:

```
> sq <- sum(a^2)
> sq
[1] 129
```

Die Funktion **sum()** addiert die Elemente eines Vektors (siehe Abschnitt 8.1.1.2).

5.3.8.4 Blockanweisung

Speziell bei bedingten Anweisungen oder Wiederholungsanweisungen ist es oft nützlich, aus mehreren Einzelanweisungen eine Blockanweisung zu erstellen. Der gesamte Block ist durch ein Paar geschweifter Klammern zu begrenzen, z.B.:

```
> if(a>0) {b<-log(a); c <- b+a}
```

5.4 Mit Skripten arbeiten

Statt mehrere zusammengehörige Anweisungen Zeile für Zeile in der **R**-Bedienoberfläche RGui abzuschicken, erstellt man besser ein **R-Skript**, was im RGui nach dem Menübefehl

Datei > Neues Skript

über einen integrierten Editor unterstützt wird:

Namenlos - R Editor

```
# Summe der quadrierten natürlichen Zahlen von 1 bis 10
s <- 0
for(i in 1:10) s = s + i^2
s
```

Um das Skript *komplett* ausführen zu lassen, kann man den Menübefehl

Bearbeiten > Alles ausführen

verwenden oder ...

- den Text vollständig markieren (z.B. über die Tastenkombination **Strg+A**)
- und dann den Schalter betätigen oder die Tastenkombination **Strg+R** verwenden.

Im Beispiel erscheint in der R-Konsole das Ergebnis:

```
[1] 385
```

Um ein Skript *partiell* ausführen zu lassen, ...

- markiert man die gewünschten Kommandos wie im folgenden Beispiel

```
Namenlos - R Editor
# Summe der quadrierten natürlichen Zahlen von 1 bis 10
s <- 0
for(i in 1:10) s = s + i^2
s
```

- und veranlasst die Ausführung mit dem Schalter oder der Tastenkombination **Strg+R**.

Mit dem Skripteditor entstehen schnell wiederverwendbare Anweisungsfolgen, die über den Menübefehl

Datei > Speichern unter

in eine Datei mit der Namenserweiterung **.R** gespeichert werden sollten.

Um ein vorhandenes Skript zu nutzen, kann man es über den Menübefehl

Datei > Öffne Skript

öffnen und dann wie oben beschrieben ausführen lassen.

Oft ist es sinnvoller, ein Skript über die **source()**-Funktion auszuführen, wobei im Skript definierte Daten- und Funktionsobjekte angelegt werden. Wenn sich die Skript-Datei nicht im Arbeitsverzeichnis befindet, ist der komplette Pfadname anzugeben, wobei die Pfadbestandteile unter Windows durch einen einfachen Vorwärtsschrägstrich oder einen doppelten Rückwärts-Schrägstrich zu trennen sind, z.B.:

```
> source("u:/eigene dateien/r/Schleife.R")
```

Wenn in **Schleife.R** das obige Beispielprogramm gespeichert ist, bleibt der **source()** -Aufruf allerdings ohne Ausgabe, weil in dieser Situation implizite **print()** - Aufrufe nicht klappen. Wer eine Ausgabe sehen möchte, muss entweder im Skript den impliziten **print()**-Aufruf durch einen expliziten ersetzen (vgl. Abschnitt 5.3.3.2), oder im **source()** - Aufruf ein Echo anfordern, z.B.:

```
> source("u:/eigene dateien/r/Schleife.R", echo=TRUE)
```

Trotz der Empfehlung, das direkte Abschicken von Anweisungen an der Eingabeaufforderung der **R**-Konsole eher zu vermeiden, werden auch im weiteren Verlauf des Manuskripts der besseren Unterscheidbarkeit halber die Anweisungen im Stil von Direkteingaben präsentiert (mit Prompt und in roter Farbe).

5.5 Generische Funktionen und Ausgabenverwaltung

Viele **R** – Funktionen können mit Daten unterschiedlichen Typs arbeiten und dabei das jeweils passende Verhalten zeigen. Man spricht hier von *generischen Funktionen*. Ein Beispiel ist die Funktion **summary()**, der man das Ausgabeobjekt einer Statistikprozedur übergibt, um über den meist spärlichen Standardausgabeumfang hinaus weitere Details zu erfahren. Im folgenden Beispiel wird für 3 künstliche Fälle mit der Funktion **lm()** eine Regression der Variablen `y` auf die Variable `x` angefordert, wobei die Ergebnisse im Objekt `lmod` landen. Die mit einem impliziten **print()**-Aufruf angeforderte Ausgabe beschränkt sich auf die Modellformel und die Regressionskoeffizienten (ohne Signifikanztests):

```
x <- c(1,2,3)
y <- x + rnorm(3,0,1)
lmod <- lm(y ~ x)
lmod
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     1.2394       0.6929
```

Im Beispiel stecken die Ergebnisse in einem Objekt mit der Klasse **lm**,

```
> class(lmod)
[1] "lm"
```

und die **print()** - Funktion zeigt eine für **R** typische Zurückhaltung bei der Ergebnisausgabe. Um eine ausführlichere Ausgabe zu erhalten, wendet man die **summary()** - Funktion auf das **lm**-Objekt an:

```
> summary(lmod)

Call:
lm(formula = y ~ x)

Residuals:
       1        2        3 
-0.01326  0.02651 -0.01326 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)  
(Intercept)  1.23940    0.04960   24.99   0.0255 *
x            0.69291    0.02296   30.18   0.0211 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03247 on 1 degrees of freedom
Multiple R-squared: 0.9989,     Adjusted R-squared: 0.9978 
F-statistic: 910.8 on 1 and 1 DF,  p-value: 0.02109
```

Analoge Unterschiede im Ausgabeumfang von **print()** und **summary()** zeigen sich bei der Funktion **mean()**. Der Klarheit halber ist diesmal der **print()**-Aufruf explizit notiert:

```
> maus <- mean(y)
> print(maus)
[1] 2.625219
> summary(maus)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  2.625   2.625   2.625   2.625   2.625   2.625
```

In der unterschiedlichen Behandlung der beiden Ausgabeobjekte (erzeugt per **lm()** bzw. **mean()**) zeigt sich die Generizität der **summary()**-Funktion. Im Grunde dient die **summary()**-Funktion nur dazu, aufgrund des Argumenttyps die im Hintergrund tatsächlich auszuführende Funktion zu wählen.

5.6 Eigene Funktionen

Als einfaches Beispiel für die Entwicklung von statistischen Algorithmen mit **R** soll eine Funktion namens `pCor()` erstellt werden, die für zwei numerische Vektoren `x` und `y` die folgendermaßen definierte Pearson-Korrelation berechnet:

$$r(x, y) := \frac{\mathrm{Cov}(x, y)}{\sqrt{\mathrm{Var}(x)\,\mathrm{Var}(y)}}$$

Bei der Definition kommt man dank der nützlichen Operatoren und Funktionen in **R** mit drei Anweisungen aus (siehe Abschnitt 8.1.1.2 zu den Funktionen **mean()** und **sum()**):

```
pCor <- function(x, y) {
  xz <- x - mean(x)
  yz <- y - mean(y)
  c(cor = sum(xz*yz) / (sum(xz^2) * sum(yz^2))^0.5)
}
```

Dabei wird allerdings die bequeme und leider oft unrealistische Annahme gemacht, dass alle Werte vorhanden sind.

Auf das Schlüsselwort **function** folgt zwischen runden Klammern eine Liste mit den formalen Argumenten. Im Rumpf einer Funktion verwendet man die formalen Argumente wie lokale Variablen, die beim Aufruf durch die übergebenen Argumente initialisiert worden sind. **R** verwendet generell Wertargumente, d.h. funktionsintern vorgenommene Änderungen haben keinen Effekt auf die aufrufende Umgebung.

In der letzten Anweisung einer Funktion legt man ihre Rückgabe fest. Im Beispiel wird (in Anlehnung an Muenchen 2011, S. 107) mit der **c()**-Funktion ein Vektor mit einem benanntem Element erstellt (vgl. Abschnitt 5.3.4.2.6), um für eine informative Ausgabe der Funktionsrückgabe zu sorgen (siehe unten).

Zur Funktionsdefinition kann man das in Abschnitt 5.4 beschriebene Skriptfenster verwenden, z.B.:

```
Namenlos - R Editor
# Funktion zur Berechnung der Pearson-Korrelation

# Zum Testen zwei Vektoren mit r(x,y) = 0,8894574
x <- c(1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1)
y <- c(2, 2, 3, 4, 4, 5, 6, 4, 3, 3, 2, 1, 2)

pCor <- function(x, y) {
  xz <- x - mean(x)
  yz <- y - mean(y)
  c(cor = sum(xz*yz) / (sum(xz^2) * sum(yz^2))^0.5)
}
```

Nachdem das Skript ausgeführt worden ist (z.B. über den Menübefehl **Bearbeiten > Alles ausführen**), hat **R** die Funktionsdefinition zur Kenntnis genommen, und es ist ein Objekt von Typ **function** vorhanden:

```
> class(pCor)
[1] "function"
```

Zum Testen verwenden wir die folgenden Vektoren mit der bekannten Korrelation r = 0,8894574:

```
x <- c(1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1)
y <- c(2, 2, 3, 4, 4, 5, 6, 4, 3, 3, 2, 1, 2)
```

Die Funktion arbeitet offenbar korrekt:

```
> pCor(x, y)
      cor
0.8894574
```

Wenn eine Funktion ein Objekt ausgeben soll, muss die **print()**-Funktion explizit ausgerufen werden, weil implizite **print()** - Aufrufe in Funktionen nicht klappen (vgl. Abschnitt 5.3.3.2).

Abgespeicherte Funktionen lassen sich nach dem Öffnen der Skriptdatei wiederverwenden (vgl. Abschnitt 5.4).

Oft ist es sinnvoller, ein Skript über die **source()**-Funktion einzulesen, wobei im Skript definierte Funktionsobjekte angelegt werden. Wenn sich die Skript-Datei nicht im Arbeitsverzeichnis befindet, ist der komplette Pfadname anzugeben, wobei die Pfadbestandteile unter Windows durch einen einfachen Vorwärtsschrägstrich oder einen doppelten Rückwärtsschrägstrich zu trennen sind, z.B.:

```
> source("u:/eigene dateien/r/pcor.r")
```

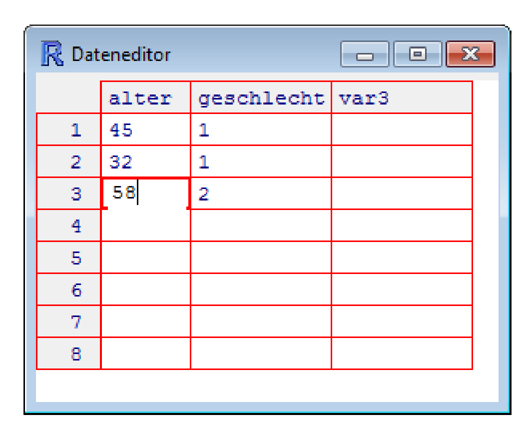
Selbstverständlich lässt sich eine selbst definierte und in einer Datei gespeicherte **R**-Funktion auch von SPSS aus nutzen. Im folgenden Beispiel wird für zwei Variablen der SPSS-Arbeitsdatei die Pearson-Korrelation mit der Funktion `pCor()` berechnet:

```
BEGIN PROGRAM R.
casedata <-spssdata.GetDataFromSPSS()
source("u:/eigene dateien/r/pCor.r")
pCor(casedata$x,casedata$y)
END PROGRAM.
```

6 Bedienungserleichterungen für R

6.1 Dateneditor

Die **R**-Bedienoberfläche hält einen einfachen Dateneditor bereit, mit dem sich Datentabellen und Matrizen anzeigen und ändern lassen:



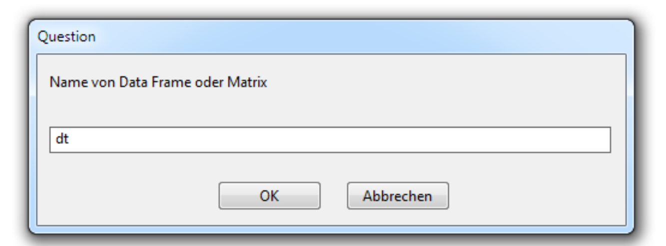
Um die Bearbeitung eines Datenobjekts zu starten, verwendet man entweder die Funktion **fix()**,

```
> fix(dt)
```

oder man wählt bei aktiver Konsole den Menübefehl

Bearbeiten > Dateneditor

und nennt anschließend das zu bearbeitende Datenobjekt:



Soll das Bearbeitungsergebnis in einem anderen Datenobjekt landen, wählt man die Funktion **edit()** mit Angabe des Rückgabeziels:

```
> dt2 <- edit(dt)
```

Der **fix()** - Funktion entspricht ein Aufruf der **edit()** - Funktion mit einem Ausgabeziel, das mit dem Argument übereinstimmt, z.B.:

```
> dt <- edit(dt)
```

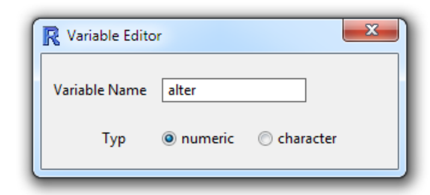
Achtung: Wenn Sie den folgenden Einstieg wählen,

```
> edit(dt)
```

wird das Bearbeitungsergebnis nach Verlassen des Editors *nicht* in `dt` gespeichert. Um die Änderungen zu retten, bleibt ihnen noch der Zugriff auf das zuletzt abgeschickte Objekt, das in **R** über den Namen **.Last.value** angesprochen werden kann (Muenchen 2011, S. 117):

```
> dt <- .Last.value
```

Mit dem Editor kann man nicht nur Daten anzeigen und editieren, sondern auch Variablen deklarieren. Den folgenden Dialog mit dem Namen und dem Datentyp zu einer Variablen erreicht man per Mausklick auf ihre Spaltenbeschriftung:



Hinweise zum Editieren:

- Aktivieren Sie nötigenfalls die Zelle zur ersten Variablen des ersten Falles, und tippen Sie den zugehörigen Wert ein.
- Drücken Sie die Tabulatortaste oder die Taste mit dem Rechtspfeil, um den eingetippten Wert zu quittieren und die Zellenmarkierung um eine Spalte nach *rechts* zu verschieben (zur nächsten Variablen).
- Auch die **Enter**-Taste quittiert den eingetippten Wert, bewegt jedoch anschließend die Zellenmarkierung um eine Zeile nach *unten* (zum nächsten Fall).
- Verwenden Sie den Punkt als Dezimaltrennzeichen.
- Wenn ein Wert fehlt, lassen Sie die betroffene Zelle einfach leer. Dort erscheint später die Anzeige **NA** (Not Available, vgl. Abschnitt 5.3.5).
- Um einen fehlerhaften Wert zu ersetzen, tragen Sie nach dem Markieren der Zelle den korrekten Wert ein.
- Um eine Eintragung zu verändern, starten Sie nach einem Doppelklick auf die betroffene Zelle das Editieren.
- Es ist *nicht* möglich, eine Änderung rückgängig zu machen.
- Eine Zeile (einen Fall) zu löschen, ist mir per Dateneditor nicht gelungen. Per Syntax gelingt es z.B. folgendermaßen, die Zeile 7 aus der Datentabelle `dt` zu löschen (vgl. Abschnitt 5.3.6):
  ```
  > dt <- dt[-7,]
  ```

Um den Editor zu beenden, können Sie ...

- den Menübefehl **Datei > Schließe** benutzen
- oder auf das Schließkreuz in der Titelzeile des Fensters klicken.

Es gibt kein Bedienelement zum Speichern. Wenn Sie das Fenster des Dateneditors schließen, wird das Ergebnis Ihrer Arbeit in das beim Editorstart vereinbarte Zielobjekt (im flüchtigen Hauptspeicher!) übertragen. Damit ist der Dateneditor für umfangreiche manuelle Dateneingaben bzw. -modifikationen wenig geeignet, denn zum Zwischenspeichern an einen sicheren Ort muss man ...

- den Dateneditor schließen,
- das Datenobjekt auf einen nichtflüchtigen Speicher (z.B. eine Festplatte) sichern, z.B. mit der **save()**-Funktion (siehe Abschnitt 5.1.2),
- den Dateneditor erneut öffnen.

Als Alternativen bieten sich an:

- bei sehr großen Datenmengen ein Datenerfassungsspezialist wie Data Collection Data Entry aus der SPSS-Familie
- der SPSS-Dateneditor
- ein Tabellenkalkulationsprogramm wie z.B. Excel aus dem Office-Paket von Microsoft oder Calc aus dem freien Paket Open- bzw. LibreOffice.

Wer zur Datenerfassung ein Tabellenkalkulationsprogramm verwendet, sollte nominalskalierte Merkmale (z.B. Geschlecht) numerisch kodieren (z.B. 1 = Frau, 2 = Mann) und später in **R** die explizite Wandlung in einen Faktor vornehmen (Field 2012, S. 97).

6.2 R Commander

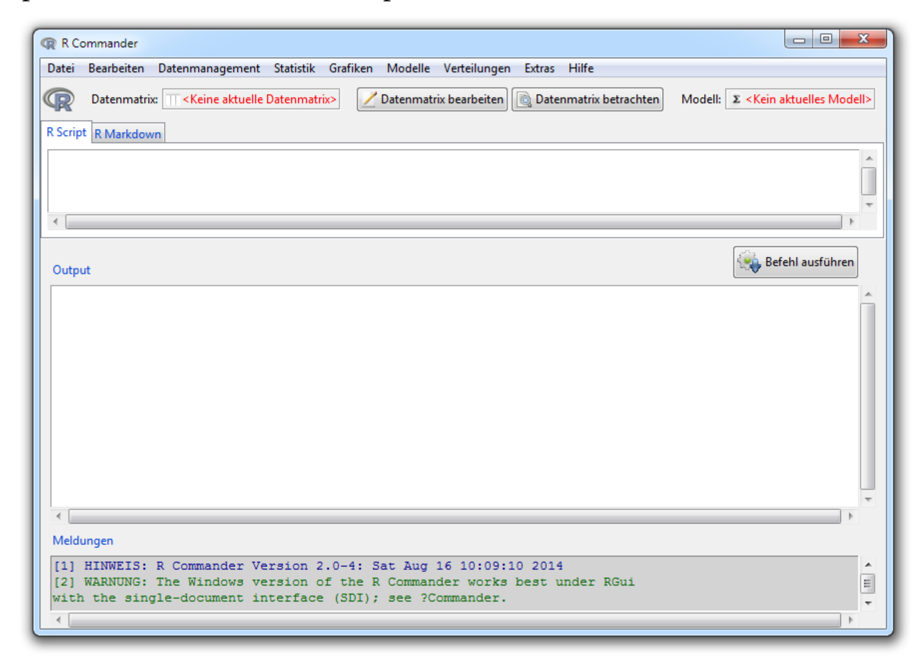
Der von John Fox (2005) entwickelte R Commander realisiert in **R** partiell die von SPSS gewohnte Bedienungsbequemlichkeit. Um Ihn nutzen zu können, muss zunächst das Paket **Rcmdr** installiert werden. Während bei **R** - Paketinstallationen die vorausgesetzten Pakete in der Regel automatisch mitinstalliert werden, ist bei **Rcmdr** aufgrund der Vielzahl von benötigten Paketen die explizite Aufforderung zur Auflösung der Abhängigkeiten durch den Wert **TRUE** für das Argument **dependencies** nach wie vor erforderlich:

```
> install.packages("Rcmdr", dependencies= TRUE)
```

Zum Starten von **Rcmdr** lädt man das Paket wie gewohnt mit der **library()** - Funktion:

```
> library(Rcmdr)
```

Es erscheint ein separates Fenster mit viel versprechenden Menüitems:



In diesem Abschnitt werden wir den R Commander kennen lernen und zu einfachen Datenverwaltungsarbeiten verwenden. Im weiteren Verlauf des Manuskripts kommt das Programm immer wieder bei Datentransformationen und -auswertungen zum Einsatz. Weiterführende Erläuterungen zum R Commander bietet z.B. ein im Internet kostenlos verfügbares Manuskript von Fox & Bouchet-Valat (2013).

6.2.1 Datentabelle anlegen, definieren und füllen

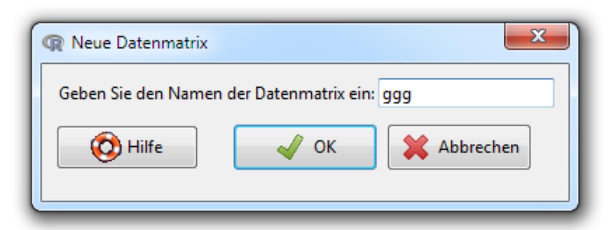
Wir werden anschließend die folgenden Daten zum Ernährungsverhalten einer studentischen Stichprobe in eine Datentabelle eintragen:

Geschlecht	Größe (in cm)	Gewicht (in kg)
2	186	82
2	178	72
2	182	75,5
1	160	65
1	168	66
1	unbekannt	76
1	165	55
2	179	76,5
1	158	50,5
2	175	80
1	176	62
2	176	unbekannt

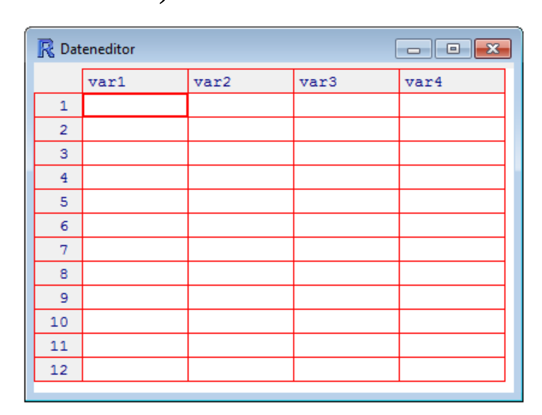
Starten Sie im Commander mit dem Menübefehl

Datenmanagement > Neue Datenmatrix

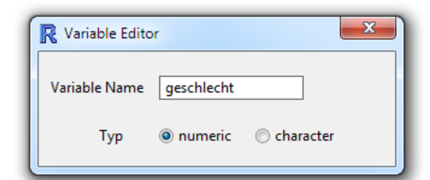
die Definition einer neuen Datentabelle, und tragen Sie im folgenden Dialog den gewünschten Namen ein, z.B.:[1]



Anschließend kann der Commander vorübergehend keine Eingaben entgegennehmen, und der Mauszeiger wird über dem Commander-Fenster zum Warten-Signal . In der **R**-Konsole erscheint hingegen ein Fenster des Dateneditors (vgl. Abschnitt 6.1):



Legen Sie für drei Variablen den numerischen Datentyp und passende Namen fest, indem Sie jeweils per Mausklick auf die Spaltenüberschrift den folgenden Dialog öffnen,

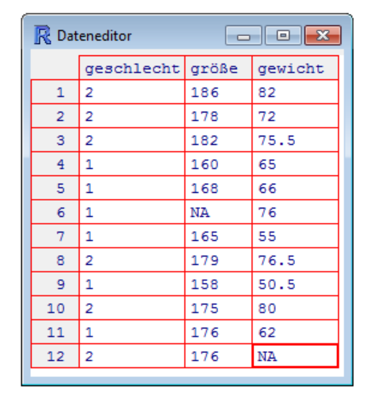


[1] Im Manuskript wird der Name ggg verwendet, weil die drei Merkmale (Geschlecht, Größe, Gewicht) zufälligerweise alle mit dem Buchstaben *G* beginnen:

für passende Werte sorgen und den Dialog dann per Schließkreuz beenden:

	geschlecht	größe	gewicht
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

Geben Sie nun die Daten ein, und beachten Sie dabei die Hinweise in Abschnitt 6.1:

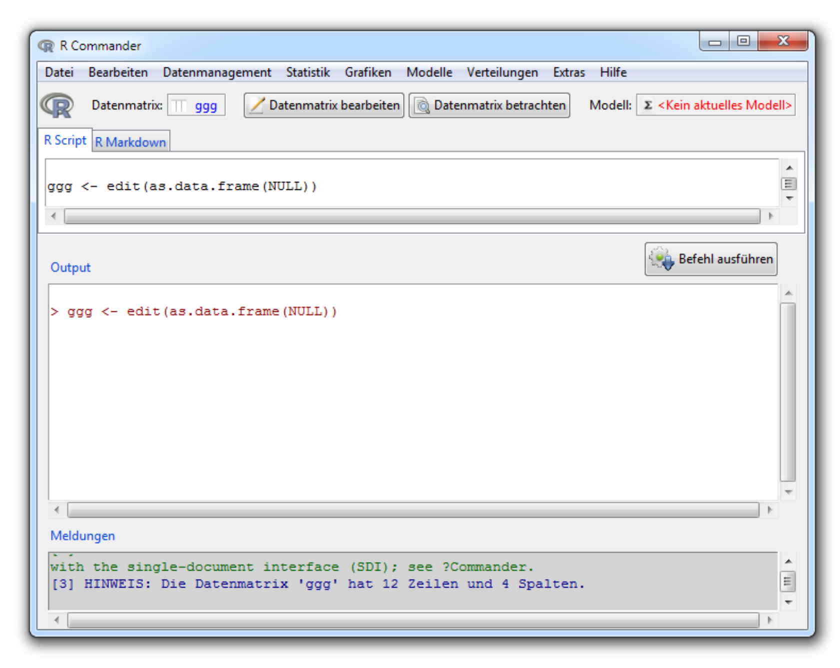


	geschlecht	größe	gewicht
1	2	186	82
2	2	178	72
3	2	182	75.5
4	1	160	65
5	1	168	66
6	1	NA	76
7	1	165	55
8	2	179	76.5
9	1	158	50.5
10	2	175	80
11	1	176	62
12	2	176	NA

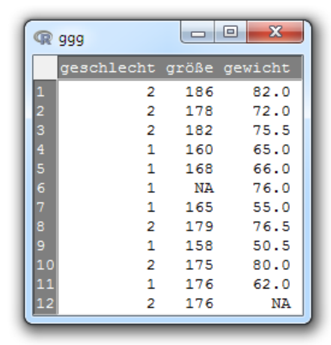
Zur Anpassung der Spaltenbreiten können Sie ...

- im Bereich der Spaltenbeschriftung die rechten Spaltengrenzen per Maus packen und verschieben
- oder nach einem rechten Mausklick auf die Tabelle aus dem Kontextmenü das Item **Autosize Spalte** wählen.

Wenn Sie die Dateneingabe beenden (z.B. mit dem Menübefehl **Datei > Schließe**) zeigt der R Commander in der **Script**-Zone an, dass die **R** - Funktion **edit()** im Einsatz war:



Außerdem ist nun die **Datenmatrix** ggg eingestellt. Man kann sie im folgenden Fenster **betrachten**



	geschlecht	größe	gewicht
1	2	186	82.0
2	2	178	72.0
3	2	182	75.5
4	1	160	65.0
5	1	168	66.0
6	1	NA	76.0
7	1	165	55.0
8	2	179	76.5
9	1	158	50.5
10	2	175	80.0
11	1	176	62.0
12	2	176	NA

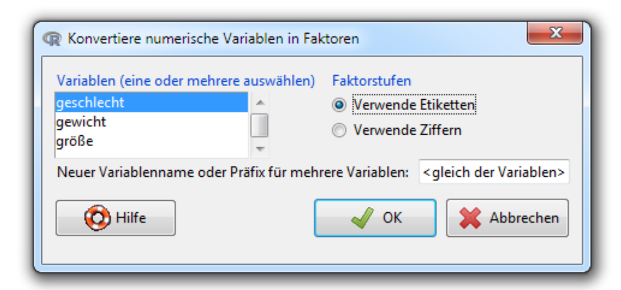
oder **bearbeiten**, wobei im letztgenannten Fall ein Dateneditorfenster in der **R**-Konsole erscheint.

6.2.2 Datenverwaltung

Als Beispiel für die Datenverwaltung mit dem R Commander wandeln wir den Vektor `geschlecht` in einen Faktor (vgl. Abschnitt 5.3.4.7.1). Nach dem Start mit dem Menübefehl

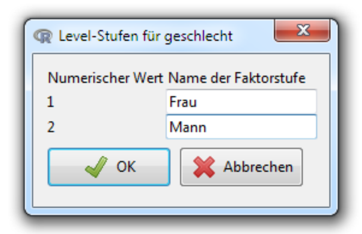
Datenmanagement > Variablen bearbeiten >
Konvertieren numerische Variablen in Faktoren

markieren wir im folgenden Dialog



die Variable `geschlecht` und geben keinen **neuen Variablennamen** an, so dass keine neue Variable entsteht, sondern die vorhandene einen neuen Typ erhält.

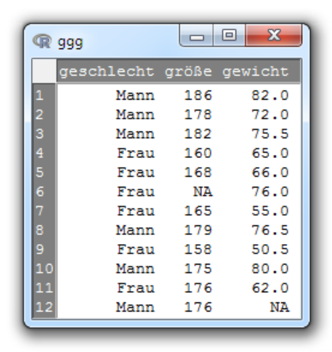
Um Wertbeschriftungen zu ermöglichen, wählen wir die **Faktorstufen**-Option **Verwende Etiketten**, so dass nach dem Quittieren mit **OK** der folgende Dialog erscheint und die Definition von Etiketten erlaubt:



Im **Script**-Fenster des Commanders wird der involvierte **factor()** - Funktionsaufruf protokolliert, der aufgrund unserer Lernerfahrungen aus Abschnitt 5.3.4.7 leicht zu verstehen ist:

```
ggg$geschlecht <- factor(ggg$geschlecht, labels=c('Frau','Mann'))
```

Wenn wir die Datentabelle erneut **betrachten**, ist das Ergebnis der Datentyp-Konvertierung zu besichtigen:

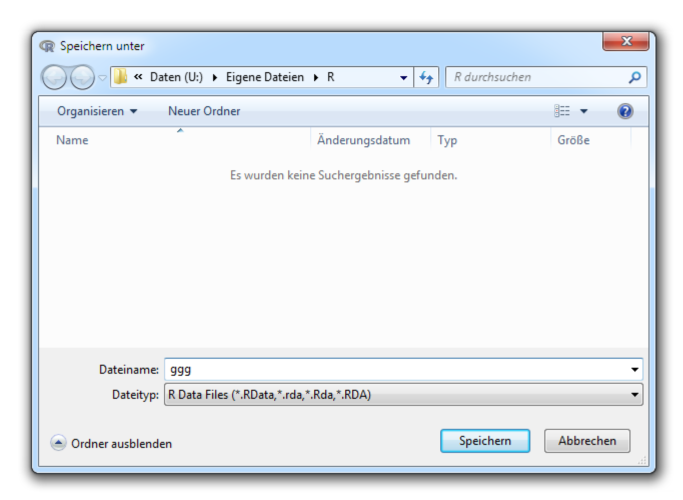


	geschlecht	größe	gewicht
1	Mann	186	82.0
2	Mann	178	72.0
3	Mann	182	75.5
4	Frau	160	65.0
5	Frau	168	66.0
6	Frau	NA	76.0
7	Frau	165	55.0
8	Mann	179	76.5
9	Frau	158	50.5
10	Mann	175	80.0
11	Frau	176	62.0
12	Mann	176	NA

Um die Datentabelle ggg in eine **RData**-Datendatei zu sichern, wählen wir den Menübefehl

Datei > Datendatei speichern unter

Wir wählen **R Data Files** als **Dateityp** und geben keine Namenserweiterung an, so dass eine Datei mit dem Namen **ggg.RData** entsteht:



Wir beenden den Commander und verzichten darauf, das Skript und weitere Commander-Produktionen zu speichern.

7 Datenverwaltung und -transformation mit R

Mit der Datenverwaltung und -transformation in **R** (siehe z.B. Hain 2011, Kap. 3 und 4 oder Muenchen 2011, Kap. 10) werden wir uns aus Zeitgründen nur knapp beschäftigen. Typische und häufig bis regelmäßig anfallende Datentransformationen (z.B. Variablen berechnen oder rekodieren) sind bequemer in SPSS Statistics mit der Standard-Syntax oder Dialogboxen zu erledigen.

Bei komplexen Operationen, die mit der regulären SPSS-Syntax kaum zu realisieren sind (z.B. Implementierung von Algorithmen), haben SPSS-Anwender die Wahl zwischen zahlreichen Programmieroptionen. Die in SPSS integrierte Programmiersprache **MATRIX** zu nutzen, hat den Vorteil, dass die erarbeiteten Lösungen von anderen SPSS-Anwendern direkt (ohne Installation von Zusatzkomponenten) übernommen werden können. Bei Verwendung von **R** besteht eine größere Wahrscheinlichkeit, dass man auf vorhandene Teil- oder Fertiglösungen zurückgreifen kann. Wird ein **R**-Skript erstellt, sind dort natürlich auch gewöhnliche Datentransformationen zu erledigen. Daher werden in diesem Abschnitt nach einer Erläuterung des lesenden und schreibenden Zugriffs auf Datendateien auch elementare Datentransformationen mit **R** behandelt (z.B. Berechnung neuer Variablen, Verwendung von Zufallszahlen, Fallauswahl).

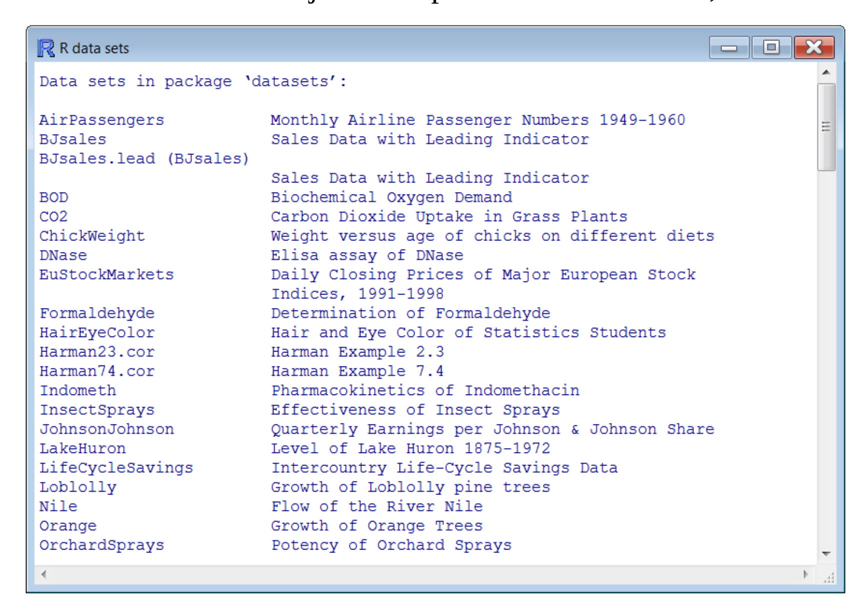
Als weitere, im Kurs nicht behandelte Programmieroptionen unterstützt SPSS noch Python, Java und die im Windows-Bereich populären .NET-Sprachen.

7.1 Beispieldaten in R-Paketen nutzen

R-Pakete enthalten oft illustrierende Beispieldaten, und diese sind sehr einfach zu nutzen. Über den Funktionsaufruf

```
> data()
```

erhält man eine Liste mit allen Datensätzen, die in aktuell geladenen Paketen verfügbar sind. In der Ausgabe sind die Pakete und deren Datensätze jeweils alphanumerisch sortiert, z.B.:


R data sets

```
Data sets in package 'datasets':

AirPassengers           Monthly Airline Passenger Numbers 1949-1960
BJsales                 Sales Data with Leading Indicator
BJsales.lead (BJsales)
                        Sales Data with Leading Indicator
BOD                     Biochemical Oxygen Demand
CO2                     Carbon Dioxide Uptake in Grass Plants
ChickWeight             Weight versus age of chicks on different diets
DNase                   Elisa assay of DNase
EuStockMarkets          Daily Closing Prices of Major European Stock
                        Indices, 1991-1998
Formaldehyde            Determination of Formaldehyde
HairEyeColor            Hair and Eye Color of Statistics Students
Harman23.cor            Harman Example 2.3
Harman74.cor            Harman Example 7.4
Indometh                Pharmacokinetics of Indomethacin
InsectSprays            Effectiveness of Insect Sprays
JohnsonJohnson          Quarterly Earnings per Johnson & Johnson Share
LakeHuron               Level of Lake Huron 1875-1972
LifeCycleSavings        Intercountry Life-Cycle Savings Data
Loblolly                Growth of Loblolly pine trees
Nile                    Flow of the River Nile
Orange                  Growth of Orange Trees
OrchardSprays           Potency of Orchard Sprays
```

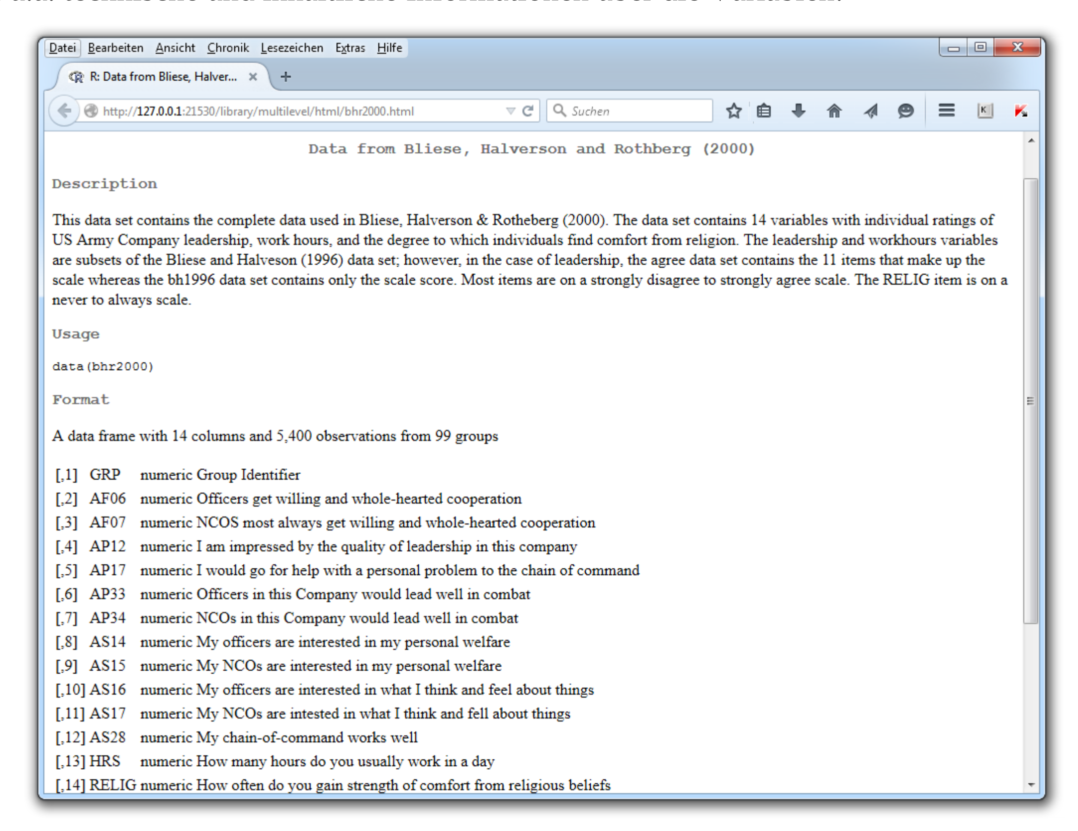
Interessiert man sich für die Datensätze in einem bestimmten Paket, kann man wie im folgenden Beispiel vorgehen:

```
> library(multilevel)
> data(package="multilevel")
```

An die zur Nutzung eines Datensatzes erforderlichen Informationen kommt man mit der **help()** - Funktion heran, z.B.:

```
> help(bhr2000)
```

Man erhält u.a. technische und inhaltliche Informationen über die Variablen:



Data from Bliese, Halverson and Rothberg (2000)

Description

This data set contains the complete data used in Bliese, Halverson & Rotheberg (2000). The data set contains 14 variables with individual ratings of US Army Company leadership, work hours, and the degree to which individuals find comfort from religion. The leadership and workhours variables are subsets of the Bliese and Halveson (1996) data set; however, in the case of leadership, the agree data set contains the 11 items that make up the scale whereas the bh1996 data set contains only the scale score. Most items are on a strongly disagree to strongly agree scale. The RELIG item is on a never to always scale.

Usage

data(bhr2000)

Format

A data frame with 14 columns and 5,400 observations from 99 groups

[,1] GRP numeric Group Identifier
[,2] AF06 numeric Officers get willing and whole-hearted cooperation
[,3] AF07 numeric NCOS most always get willing and whole-hearted cooperation
[,4] AP12 numeric I am impressed by the quality of leadership in this company
[,5] AP17 numeric I would go for help with a personal problem to the chain of command
[,6] AP33 numeric Officers in this Company would lead well in combat
[,7] AP34 numeric NCOs in this Company would lead well in combat
[,8] AS14 numeric My officers are interested in my personal welfare
[,9] AS15 numeric My NCOs are interested in my personal welfare
[,10] AS16 numeric My officers are interested in what I think and feel about things
[,11] AS17 numeric My NCOs are intested in what I think and fell about things
[,12] AS28 numeric My chain-of-command works well
[,13] HRS numeric How many hours do you usually work in a day
[,14] RELIG numeric How often do you gain strength of comfort from religious beliefs

Um ein Datenobjekt analysieren zu können, befördert man es mit einem **data()** - Aufruf in den Workspace, z.B.:

```
> data(bhr2000)
```

7.2 Daten in Fremdformaten lesen und schreiben

7.2.1 Textdateien mit separierten Daten

Eine Textdatei mit separierten Daten enthält pro Fall eine Zeile, in der alle Werte in einer festen Reihenfolge stehen, wobei sich zwischen zwei Werten ein Trennzeichen befindet. In Abhängigkeit vom Separatorzeichen unterscheidet man die Typen:

- Tabulator-separierte Textdatei
  Typische Namenserweiterungen sind **.dat** (beim Erstellen durch SPSS) und **.txt** (beim Erstellen durch Excel).
- Komma-separierte Textdatei
  Trotz der Typbezeichnung verwenden SPSS und Excel beim Erstellen einer solchen Datei (zumindest in unseren Landen) ein Semikolon als Trennzeichen, weil das Komma als Dezimaltrennzeichen benötigt wird. Als Namenserweiterung wird einheitlich **.csv** verwendet.

Eine Datei mit separierten Daten enthält oft in der ersten Zeile die Variablennamen, so dass z.B. die in Abschnitt 6.2.1 vorgestellten Daten in einer Tabulator-getrennten Textdatei so aussehen:

```
geschlecht größe     gewicht
2          186       82
2          178       72
2          182       75,5
1          160       65
1          168       66
1                    76
1          165       55
2          179       76,5
1          158       50,5
2          175       80
1          176       62
2          176
```

Fehlt bei einem Fall eine Variablenausprägung, bleibt die betroffene Zelle leer.

7.2.1.1 Lesen

Um diese Daten in eine Datentabelle einzulesen, eignet sich in **R** die Funktion **read.delim2()**, die im Unterschied zu **read.delim()** das *Komma* als Dezimaltrennzeichen interpretiert. Im folgenden Beispiel wird mit dem **header**-Argument die Anwesenheit einer einleitenden Zeile mit Variablennamen bekannt gegeben:

```
> ggg <- read.delim2("U:/Eigene Dateien/R/ggg.dat", header = TRUE)
> ggg
```

Im Ergebnis

```
   ï..geschlecht grÃ.ÃŸe gewicht
1              2    186    82.0
2              2    178    72.0
3              2    182    75.5
4              1    160    65.0
5              1    168    66.0
6              1     NA    76.0
7              1    165    55.0
8              2    179    76.5
9              1    158    50.5
10             2    175    80.0
11             1    176    62.0
12             2    176      NA
```

zeigen sich zwei Macken:

- Am Dateianfang sind unerwartete Zeichen erkannt und dem ersten Variablennamen zugeschlagen worden (mit dem Ergebnis: `ï..geschlecht`).
- Im Variablennamen `größe` sind *ö* und *ß* falsch erkannt worden.

Die Probleme resultieren daraus, dass die Eingabedatei keine ANSI-Kodierung verwendet, sondern die modernere UTF-8 - Kodierung. Im folgenden Funktionsaufruf

```
> ggg <- read.delim2("U:/Eigene Dateien/R/ggg.dat", header = TRUE, encoding="UTF-8")
> ggg
```

wird **R** per **encoding**-Argument über die Kodierung der Eingabedatei informiert, was zumindest das Umlaute-Problem behebt:

```
   X.U.FEFF.geschlecht größe gewicht
1                    2   186    82.0
2                    2   178    72.0
.                    .     .       .
```

Der verunstaltete Name der ersten Variablen resultiert aus der falsch verstandenen BOM-Sequenz (*Byte Order Mark*), die viele unter Windows erstellte UTF-8 - Dateien am Dateianfang enthalten. Mit der folgenden, leider erst ab **R** 3.0 unterstützten, Variante

```
> ggg <- read.delim2("U:/Eigene Dateien/R/ggg.dat", header = TRUE, fileEncoding="UTF-8-BOM")
> ggg
```

gelingt ein korrekter Datenimport:

```
   geschlecht größe gewicht
1           2   186    82.0
2           2   178    72.0
.           .     .       .
```

Die Fälle erhalten automatisch fortlaufende Nummern (im Beispiel von 1 bis 12). Gelegentlich sind Datendateien einzulesen, die als erste Variable eine Fallidentifikation enthalten, die nicht unbedingt eine mit 1 beginnende, lückenlose Nummerierung enthält, In der folgenden Variante

```
nr          geschlecht größe       gewicht
1           2          186         82
2           2          178         72
.           .          .           .
```

der obigen Beispieldatei ist eine einleitende Variable namens `nr` vorhanden, wobei der letzte Fall (vermutlich aus gutem Grund) den Wert 13 besitzt. In einer solchen Situation sollte man den **R**-Textdatenimport dazu überreden, die erste Variable zur Fallidentifikation zu verwenden und auf eine automatische Nummerierung zu verzichten. Dazu kann man ...

- den Namen für diese Eingabevariable aus der Datei löschen
- oder die Importfunktion mit dem Argument **row.names** darüber informieren, in welcher Eingabespalte sich die Fallbezeichnungen befinden.

So

```
> setwd("U:/Eigene Dateien/R")
> ggg <- read.delim2("gggNr.dat", header = TRUE, fileEncoding="UTF-8-BOM", row.names="nr")
> ggg
```

erhalten wir das angestrebte Ergebnis:

```
   geschlecht größe gewicht
1           2   186    82.0
2           2   178    72.0
.           .     .       .
13          2   176      NA
```

Beim Lesen der CSV-Variante der Eingabedaten (inkl. Fallidentifikationsvariable `nr`)

```
nr;geschlecht;größe;gewicht
1;2;186;82
2;2;178;72
3;2;182;75,5
4;1;160;65
5;1;168;66
6;1; ;76
7;1;165;55
8;2;179;76,5
9;1;158;50,5
10;2;175;80
11;1;176;62
13;2;176;
```

ersparen wir uns den UTF-8 - Ärger, verwenden also eine Textdatei mit ANSI-Kodierung. Als **R**-Funktion verwenden wir **read.csv2()**, wobei die 2 am Ende des Funktionsnamens signalisiert, dass ein Komma als Dezimaltrennzeichen und ein Semikolon als Separatorzeichen erwartete werden. Im folgen-

den Kommando verwenden wir unser Wissen über die Zeilenbezeichnungsvariable und kommen gleich im ersten Versuch

```
> ggg <- read.csv2("U:/Eigene Dateien/R/gggNr.csv", header = TRUE, row.names="nr")
> ggg
```

zum gewünschten Ergebnis:

```
   geschlecht größe gewicht
1           2   186    82.0
2           2   178    72.0
.           .     .       .
13          2   176      NA
```

Die im aktuellen Abschnitt vorgestellten Funktionen zum Lesen von Textdateien stellen Aufrufvereinfachungen für die Funktion **read.table()** dar, die jeweils im Hintergrund verwendet wird. Der folgenden Tabelle nach Hain (2011, S. 67) ist zu entnehmen, welche Einstellungen bzgl.

- der Einführungszeile mit Variablennamen (Argument **header**)
- des Separatorzeichens (Argument **sep**)
- und des Dezimaltrennzeichens (Argument **dec**)

mit den verschiedenen Funktionen verbunden sind ("\t" steht für das Tabulatorzeichen):

Funktion	**header**	**sep**	**dec**
read.table()	FALSE	""	"."
read.csv()	TRUE	","	"."
read.csv2()	TRUE	";"	","
read.delim()	TRUE	"\t"	"."
read.delim2()	TRUE	"\t"	","

Weil das Merkmal Geschlecht in den Beispieldateien numerisch kodiert war, ist beim Einlesen ein numerischer *Vektor* entstanden, der durch die folgende Anweisung in einen *Faktor* gewandet werden sollte:

```
> ggg$geschlecht <- factor(ggg$geschlecht, labels=c('Frau','Mann'))
> ggg
```

Das Ergebnis:

```
   geschlecht größe gewicht
1        Mann   186    82.0
2        Mann   178    72.0
.           .     .       .
13       Mann   176      NA
```

7.2.1.2 Schreiben

Zum Zweck der Kooperation mit anderen Programmen ist es oft sinnvoll, die Variablen einer **R**-Datentabelle in eine Textdatei mit separierten Daten zu befördern, weil dieses Dateiformat von praktisch jeder Statistik-Software gelesen werden kann. Zuständig ist die **R**-Funktion **write.table()**, die als erstes Argument den Namen der Datentabelle und als zweites Argument den Namen der Zieldatei erwartet. Häufig werden außerdem die Argumente mit den folgenden Namen benötigt:

- **sep**
  Zwischen Anführungszeichen kann das gewünschte Trennzeichen angegeben werden, wobei insbesondere in Frage kommen:
  - **\t**
    Tabulatorzeichen
  - **,** oder **;**
    Um eine CSV-Datei zu erzielen, verwendet man in Abhängigkeit vom Dezimaltrennzeichen zum Separieren das Komma oder das Semikolon.
- **dec**
  Per Voreinstellung verwendet **R** in der Ausgabedatei den Punkt als Dezimaltrennzeichen. Ist das Komma gewünscht, setzt man den Parameter **dec** auf den Wert ",".
- **na**
  Bei fehlenden Werten schreibt **R** per Voreinstellung **NA** in die Exportdatei. Über das Argumetn **na** lässt sich ein alternativer Ersatzwert vereinbaren (z.B. eine leere Zeichenfolge durch "").
- **row.names**
  Per Voreinstellung gibt **R** die Zeilenbeschriftungen aus, verzichtet aber in der einleitenden Zeile auf einen zugehörigen Variablennamen. Beim Einlesen durch Fremdprogramme kommt es daher oft zu einer falschen Zuordnung der Variablennamen oder zu einem Fehler. Um dieses Problem zu vermeiden, verzichtet man in der Regel auf die Ausgabe der Zeilenbeschriftungen und setzt das Argument **row.names** auf den Wert **FALSE**.
- **quote**
  Per Voreinstellung begrenzt **R** die Variablennamen und die Werte von Faktoren durch Anführungszeichen. Dies lässt sich mit dem Wert **FALSE** für das Argument **quote** verhindern.

Um die Datentabelle ggg

```
> ggg
  geschlecht größe gewicht
1       Mann   186    82.0
2       Mann   178    72.0
.          .     .       .
13      Mann   176      NA
```

in eine ...

- Textdatei im aktuellen Arbeitsverzeichnis
- mit Tabulator-getrennten Daten
- und Dezimalkomma
- ohne Zeilenbeschriftungen
- mit einer leeren Zeichenfolge statt fehlender Werte
- ohne Anführungszeichen um Variablennamen und Faktorwerte

zu schreiben, eignet sich das folgende Kommando:

```
> write.table(ggg, "ggg.dat",sep="\t",dec=",",row.names=FALSE,na="",quote=FALSE)
```

Das Ergebnis:

```
geschlecht   größe   gewicht
Mann         186     82
Mann         178     72
Mann         182     75,5
Frau         160     65
Frau         168     66
Frau                 76
.            .       .
```

Unter Windows ist die Ausgabedatei ANSI-kodiert.

Bei der Ausgabe in eine ...

- Textdatei im aktuellen Arbeitsverzeichnis
- mit Semikolon-getrennten Daten und Dezimalkomma
- ohne Zeilenbeschriftungen
- mit der voreingestellten Ausgabe fehlender Werte
- ohne Anführungszeichen um Variablennamen und Faktorwerte

erspart die Funktion **write.csv2()** im Vergleich zu **write.table()** etwas Schreibarbeit, z.B.:

```
> write.csv2(ggg, "ggg.csv", row.names=FALSE, quote=FALSE)
```

Bei Faktoren schreibt **R** Platz verschwendend die Werteetiketten (labels) in die Ausgabedatei. Auf einfache Weise eine Ausgabe der internen numerischen Codes zu erreichen, ist mir nicht gelungen.

7.2.2 SPSS-Datendatei lesen

In einer aus SPSS gestarteten **R**-Sitzung kann man bequem auf die Variablen der SPSS-Arbeitsdatei zugreifen (siehe Abschnitt 4.1). **R** kommt aber auch im selbständigen an SPSS-Variablen heran, sofern sich diese in einer SPSS-Datendatei befinden. Dazu muss zunächst das Paket **foreign** geladen werden:

```
> library(foreign)
```

Es gehört zu den so genannten *recommended packages*, dürfte also in praktisch jeder **R**-Installation vorhanden sein.

Eine SPSS-Datendatei (Namenserweiterung **.SAV**) kann mit der Funktion **read.spss()** geöffnet werden, wobei für Dateien im Arbeitsverzeichnis (vgl. Abschnitt 5.1.1) keine Pfadangabe erforderlich ist, z.B.:

```
> ggg <- read.spss("ggg.sav", to.data.frame = TRUE, reencode="utf-8")
```

read.spss() liefert per Voreinstellung eine *Liste*, lässt sich aber durch den Wert **TRUE** für das Argument **to.data.frame** dazu überreden, eine Datentabelle zu erstellen.

Wurde die SAV-Datei von einer SPSS-Version ab 18 erstellt, kommt es beim Öffnen zu Warnungen, z.B.:

```
Warnmeldung:
In read.spss("ggg.sav", to.data.frame = TRUE, reencode = "utf-8") :
  ggg.sav: Unrecognized record type 7, subtype 18 encountered in system file
```

Trotz der Warnungen scheint die Datentabelle intakt zu sein:

SPSS-Dateien werden seit der SPSS-Version 21 bevorzugt im Unicode-Modus erstellt, wobei die UTF-8 -Kodierung zum Einsatz kommt. Um beim Lesen solcher Dateien z.B. eine falsche Interpretation von Umlauten in Variablennamen und Wertbeschriftungen

```
   nr geschlecht grÃ.ÃŸe gewicht
1   1       Mann     186    82.0
.   .          .       .       .
```

zu verhindern, muss das Argument **reencode** den Wert **utf-8** erhalten. Bei SPSS-Dateien in traditioneller Kodierung wirkt der Wert **urt-8** kontraproduktiv. Die korrekte Einstellung muss man durch Ausprobieren ermitteln.

Als Rückgabe liefert die Funktion **read.spss()** per Voreinstellung eine Liste (vgl. Abschnitt 5.3.4.6), was man in der Regel durch den Wert **TRUE** für das Argument **to.data.frame** verhindert.

Über das Argument **use.value.labels** mit dem Voreinstellungswert **TRUE** entscheidet man darüber, ob numerische Variablen in Faktoren konvertiert werden sollen, wenn für alle Werte ein Etikett vorhanden ist. Im Beispiel ist diese Konvertierung bei der Variablen `geschlecht` passiert.

Im Unterschied zu den Funktionen zum Lesen von separierten Textdateien (siehe Abschnitt 7.2) besitzt **read.spss()** kein Argument, um für eine vorhandene Eingabevariable die Verwendung zur Zeilenbeschriftung zu veranlassen. Dieses Ziel lässt sich aber in zwei kurzen Anweisungen doch realisieren. Zunächst werden die gewünschten Fallbeschriftungen aus der Variablen `nr` gelesen. Dann wird der Variablen `nr` der Wert **NULL** zugewiesen, um sie aus der Datentabelle zu entfernen:

```
> row.names(ggg) <- ggg$nr
> ggg$nr <- NULL
```

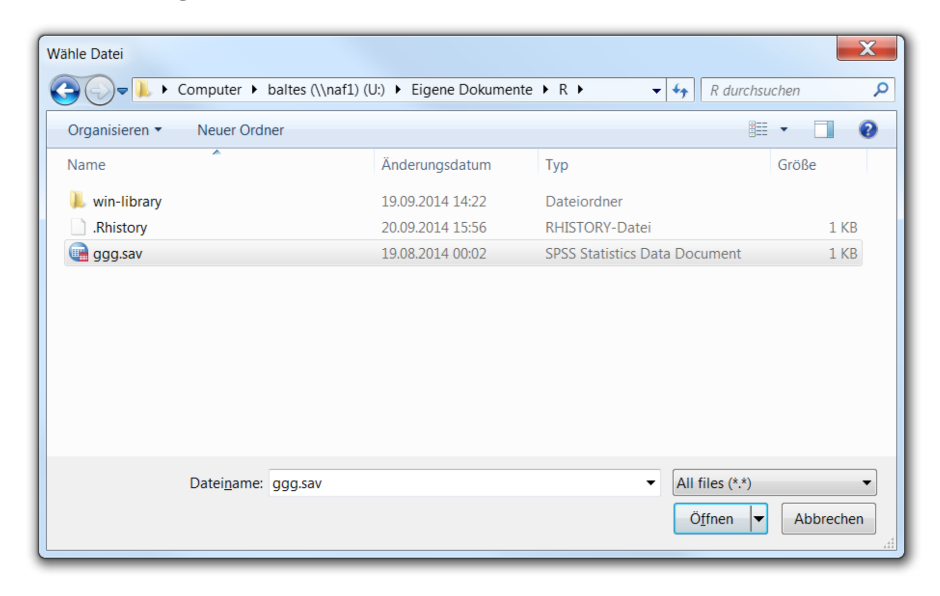
Das Ergebnis:

```
     geschlecht größe gewicht
1          Mann   186    82.0
.             .     .       .
13         Mann   176      NA
```

Eine Möglichkeit, mit **R** eine SPSS-Datendatei zu *schreiben*, ist mir nicht bekannt. Insgesamt sind für den Datenaustausch zwischen **R** und SPSS die in Abschnitt 4 beschrieben Techniken auf der Basis der R-Essentials gegenüber dem Dateitransfer zu bevorzugen.

7.2.3 Dateiauswahl per Dialogbox

Um eine Datei per GUI-Dialog wählen zu können,



verwendet man den Funktionsaufruf **file.choose()**, der überall erlaubt ist, wo in **R**-Syntax ein Dateiname erwartet wird, z.B.:

```
> ggg <- read.spss(file.choose(),to.data.frame = TRUE, reencode="utf-8")
```

7.3 Variablen berechnen oder modifizieren

7.3.1 Umkodieren

Das in der empirischen Forschungspraxis häufig erforderliche Umkodieren von Werten lässt sich z.B. mit Hilfe von logischen Indexvektoren (vgl. Abschnitt 5.3.6.3) erledigen. Im folgenden Beispiel werden nach einem von Wollschläger (2010, S. 39) vorgeschlagenen Verfahren die Werte eines Vektors folgendermaßen umkodiert:

$$5 \rightarrow 1$$
$$4 \rightarrow 2$$
$$3 \rightarrow 3$$
$$2 \rightarrow 4$$
$$1 \rightarrow 5$$

Es ist in der Regel sinnvoll (und im gleich beschriebenen Verfahren auch von praktischem Vorteil), den Ausgangsvektor unverändert zu lassen und einen neuen Vektor mit den umkodierten Werten zu erstellen:

```
> v <- c(2,NA,4,1,5,2,4,3)
> w <- numeric(length(v))
> w[v==1] <- 5
> w[v==2] <- 4
> w[v==3] <- 3
> w[v==4] <- 2
> w[v==5] <- 1
> w[w==0] <- NA
> w
[1] 4 NA 2 5 1 4 2 3
```

Mit der Funktion **numeric()** (siehe Abschnitt 5.3.4.2.1) wird ein neuer numerischer Vektor passender Länge erstellt, dessen Elemente alle mit 0 initialisiert sind. Dann werden für jeden möglichen Wert des Ausgangsvektors über einen logischen Indexvektor die passenden Elemente des Zielvektors bestimmt und auf den gewünschten Wert gesetzt. **NA**-Werte des Ausgangsvektors werden bei dieser Prozedur auf die 0 abgebildet (= initialer Wert des Zielvektors). Durch die abschließende Zuweisung

```
> w[w==0] <- NA
```

erhalten diese Elemente den Wert **NA**.

Soll das Umkodieren zu einem Vektor mit lediglich zwei möglichen Werten führen, eignet sich die **ifelse()**-Funktion, z.B.:

```
> w <- ifelse(test=(v>3), yes=1, no=0)
> w
[1]  0 NA  1  0  1  0  1  0
```

Ein logischer Vektor fungiert als **test**-Argument, und die Funktion **ifelse()** liefert einen Vektor entsprechender Länge als Ergebnis. Die Elemente dieses Ergebnisvektors sind …

- identisch mit dem **yes**-Argument, wenn das korrespondierende **test**-Element gleich **TRUE** ist,
- identisch mit dem **no**-Argument, wenn das korrespondierende **test**-Element gleich **FALSE** ist
- und gleich **NA**, wenn das **test**-Element gleich **NA** ist.

Als **yes**- bzw. **no**-Argument sind auch Vektoren erlaubt. In diesem Fall ist das *i*-te Element des Ergebnisvektors identisch mit dem *i*-ten Element des **yes**-Vektors, wenn der *i*-te Wahrheitswert im **test**-Vektor gleich **TRUE** ist, und gleich dem *i*-ten Element des **no**-Vektors, wenn der *i*-te Wahrheitswert gleich **FALSE** ist.

Mit der Funktion **cut()** lässt sich ein numerischer Vektor über eine Liste von Aufteilungspunkten in einen Faktor wandeln. Im folgenden Beispiel enthält der Ausgangsvektor standardnormalverteilte Zufallszahlen, die von der Funktion **rnorm()** erstellt werden (siehe Abschnitt 7.4.1.1):

```
> numvec <- rnorm(100, 0, 1)
> fac <- cut(numvec, breaks=c(-Inf,-2,-1,0,1,2,Inf))
> str(fac)
 Factor w/ 6 levels "(-Inf,-2]","(-2,-1]",..: 2 4 3 5 1 4 2 4 3 5 ...
> table(fac)
fac
(-Inf,-2]   (-2,-1]    (-1,0]     (0,1]     (1,2]  (2, Inf]
        2        14        34        40         8         2
```

Durch die im Argumentvektor **breaks** enthaltenen Aufteilungspunkte werden links-offene und rechts-abgeschlossene Intervalle festgelegt. Die Funktion **table()** (siehe Abschnitt 8.1.2) liefert die absoluten Häufigkeiten der Kategorien.

7.3.2 Berechnen

Einen neuen Vektor aus bereits vorhandenen Vektoren zu berechnen, ist in **R** eine leichte Übung. Als Beispiel soll aus zwei Vektoren mit der Körpergröße (in cm) bzw. mit dem Körpergewicht (in kg) von 12 Personen

```
> groesse <- c(186,178,182,160,168,NA,165,179,158,175,176,176)
> gewicht <- c(80,71,75.5,65,66,76,55,76.5,50.5,80,62,NA)
```

der *Body Mass Index* (BMI) nach folgender Formel berechnet werden:

$$\frac{Gewicht\ (in\ kg)}{Größe^2\ (in\ m)}$$

Hinweise zur Formulierung des numerischen Ausdrucks:

- Als Symbol für das Potenzieren sind zwei unmittelbar aufeinanderfolgende Sternchen (**) oder ein Dach (^) zu verwenden.
- Die Körpergröße muss von der Einheit Zentimeter in die Einheit Meter umgerechnet werden.
- Wegen der Auswertungsprioritäten der beteiligten Operatoren (in absteigender Reihenfolge: Potenzieren, Dividieren) kann z.B. der folgende numerische Ausdruck verwendet werden:

```
> bmi <- gewicht/(groesse/100)^2
```

Das Ergebnis:

```
[1] 23.12406 22.40879 22.79314 25.39062 23.38435       NA 20.20202 23.87566
[9] 20.22913 26.12245 20.01550       NA
```

Erwartungsgemäß haben die beiden Fälle mit einem fehlenden Wert bei einer Ausgangsvariablen als Berechnungsergebnis den Wert **NA** erhalten.

7.4 Zufallszahlen erzeugen

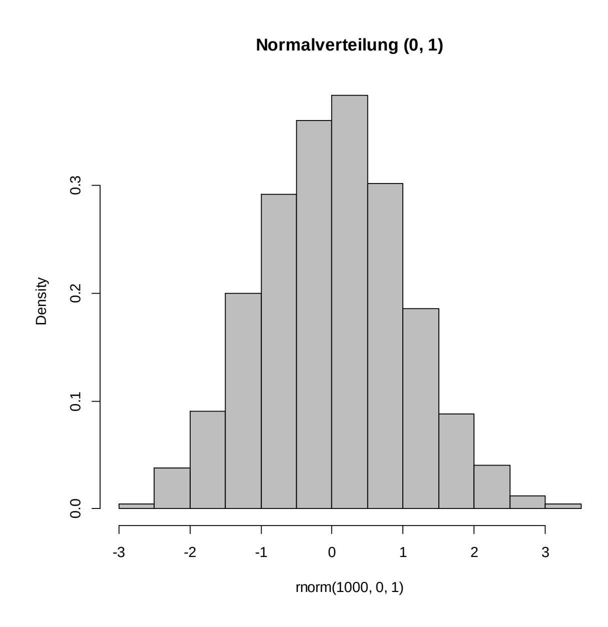
Für Simulationsstudien zu Phänomenen und Methoden der Statistik werden Stichproben aus einer Population mit bekannter Verteilung benötigt. So lässt sich etwa das Verhalten von statistischen Auswertungsprozeduren bei bekannten Verteilungsverhältnissen untersuchen. **R** enthält diverse Funktionen, um univariate Zufallsstichproben aus einer definierten Verteilung zu ziehen. Anschließend werden einige Vertreter vorgestellt.

7.4.1.1 Normalverteilte Zufallszahlen

Über die Funktion **rnorm()** erhält man *n* Zufallszahlen aus einer normalverteilten Population mit bestimmtem Erwartungswert und bestimmter Standardabweichung, z.B.:

```
> sampnor <- rnorm(10, 0, 1);sampnor
[1]  0.5901100  0.5949126  0.7150877  1.1859644  0.8376390 -0.4352961
[7] -1.3281079  0.7989749  1.4540338 -1.5676006
```

Hier ist das Histogramm für eine erheblich größere Stichprobe aus einer Standardnormalverteilung zu sehen:



Es wurde durch den folgenden Aufruf der **R**-Funktion **hist()** erstellt (vgl. Abschnitt 9.2.4.4)

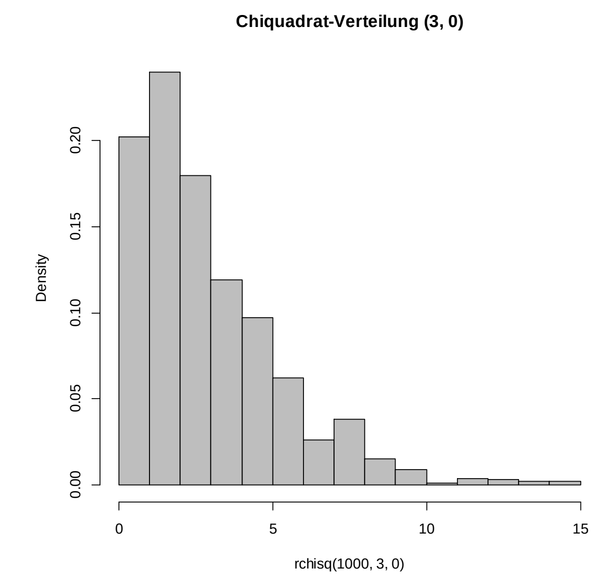
```
> hist(rnorm(1000, 0, 1), freq=FALSE, col="grey",main="Normalverteilung (0, 1)")
```

7.4.1.2 χ^2-verteilte Zufallszahlen

Über die Funktion **rchisq()** erhält man *n* Zufallszahlen aus einer χ^2-Verteilung, wobei die Anzahl der Freiheitsgrade und der Nonzentralitätsparameter einstellbar sind, z.B.:

```
> sampchisq <- rchisq(10, 3, 0);sampchisq
[1] 1.1991623 1.1538841 0.9253693 0.3756986 1.2608549 6.2716147 0.6392600
[8] 1.9859259 0.9282716 3.4587970
```

Hier ist das Histogramm für eine erheblich größere Stichprobe aus einer χ^2-Verteilung mit 3 Freiheitsgraden und Nonzentralitätsparameter 0 zu sehen:



Es wurde durch den folgenden Aufruf der **R**-Funktion **hist()** erstellt (vgl. Abschnitt 9.2.4.4)

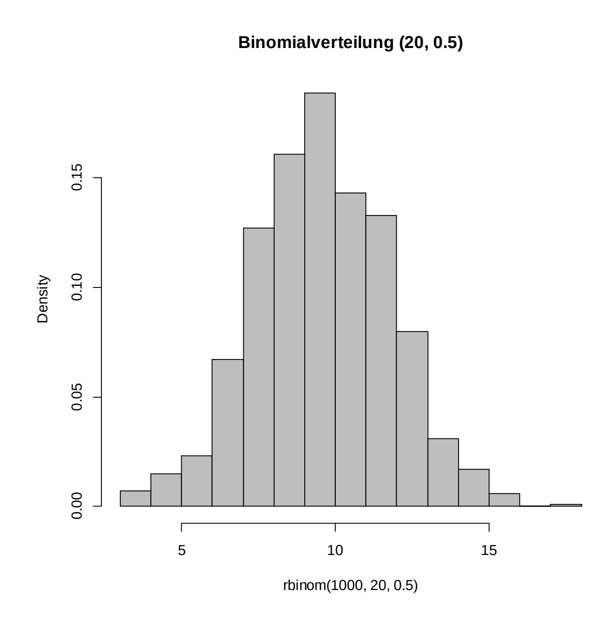
```
> hist(rchisq(1000, 3, 0), freq=FALSE, col="grey", main="Chiquadrat-Verteilung (3, 0)")
```

7.4.1.3 Binomialverteilte Zufallszahlen

Über die Funktion **rbinom()** erhält man *n* Zufallszahlen aus einer binomialverteilten Population mit bestimmten Parametern für das zugrunde liegende Bernoulli-Experiment (Anzahl der Wiederholungen, Wahrscheinlichkeit im Einzelexperiment), z.B.:

```
> sampbin <- rbinom(10, 20, 0.5);sampbin
[1] 13  8 12 11 10 12 12 10 10 10
```

Hier ist das Histogramm für eine erheblich größere Stichprobe aus derselben Binomialverteilung zu sehen:



Es wurde durch den folgenden Aufruf der **R**-Funktion **hist()** erstellt (vgl. Abschnitt 9.2.4.4)

```
> hist(rbinom(1000,20,0.5), freq=FALSE, col="grey", main="Binomialverteilung (20, 0.5)")
```

7.5 Auswahl von Fällen und/oder Variablen

7.5.1 Auswahl von Fällen

Oft soll bei einer Auswertung nur eine Teilmenge der Fälle in einer Datentabelle einbezogen werden. Mit der Funktion **subset()** lässt sich aus einer Datentabelle über einen logischen Ausdruck leicht eine Teilmenge mit allen Fällen ermitteln, die den logischen Ausdruck erfüllen, z.B.:

```
> dt <- data.frame(alter=c(45,32,58), ges=c('f','f','m'), mot=c(4,2,9))
> dtf <- subset(dt, ges=='f')
> dtf
  alter ges mot
1    45   f   4
2    32   f   2
```

Zu den im logischen Ausdruck verwendbaren relationalen und logischen Operatoren siehe Abschnitt 5.3.7.

Eine alternative Möglichkeit zur Fallauswahl bietet der in Abschnitt 5.3.6.3 beschriebene logische Indexvektor, z.B.:

```
> dtf <- dt[dt$ges=='f',]
```

Beim Indexzugriff auf eine Datentabelle muss hinter dem logischen Ausdruck für die Auswahl der Zeilen auf jeden Fall ein Komma stehen. Dann kann optional z.B. ein numerischer Vektor mit Spaltennummern zur einschränkenden Wahl der Variablen folgen (siehe Abschnitt 7.5.2).

7.5.2 Auswahl von Variablen

Bei einer großen Datentabelle kann es sich lohnen, die bei einer Auswertung tatsächlich benötigten Variablen in eine reduzierte Datentabelle zu extrahieren. Weil **R** alle geladenen Datenobjekte komplett im Hauptspeicher des Computers hält, ist bei einer großen Anzahl von Fällen das Entfernen von irrelevanten Variablen sinnvoll.

Die in Abschnitt 7.5.1 beschrieben Funktion **subset()** ermöglicht ergänzend zur bzw. anstatt einer Fallauswahl über das Argument **select** auch eine Variablenauswahl. Im ersten Beispiel findet ausschließlich eine Variablenauswahl statt, die über einen Indexvektor erfolgt:

```
> dt12 <- subset(dt, select=c(1,2))
```

Im zweiten Beispiel finden eine Fall- und eine Variablenauswahl statt. Zur Variablenauswahl dient ein logischer Ausdruck, wobei mit Hilfe der Funktion **grepl()** alle Variablen mit einer bestimmten Teilzeichenfolge im Namen ausgewählt werden:

```
> dtf <- subset(dt, ges=='f', select=grepl("e", names(dt)))
```

Auch beim Indexzugriff auf eine Datentabelle ist eine Variablenauswahl möglich, wobei die zweite Indexdimension (nach dem Komma) einen numerischen oder logischen Auswahlvektor erhält. Es folgen die beiden obigen Beispiele mit Indexsyntax:

```
> dt12 <- dt[, c(1,2)]
> dtf <- dt[dt$ges=='f', grepl("e", names(dt))]
```

7.6 Daten aus verschiedenen Tabellen zusammenführen

Wurden Daten mit einer hierarchischen Struktur erhoben (z.B. Clusterstichproben mit Beobachten zu den Individuen und den Gruppen, Längsschnittdaten mit Beobachtungen zu den Messzeitpunkten und den Subjekten), dann liegen die Daten zur Mikro- bzw. Makroebene oft in zwei getrennten Datentabellen vor.

Im folgenden Beispiel sind die Daten der Mikroebene (Leistungsmessungen in den Fächern Mathematik und Geographie für Schüler in drei Klassen) in der Datentabelle `dfInd` zu finden und die Daten der Makroebene (Größe der Klasse) in der Datentabelle `dfGr`:

```
> dfInd <- data.frame(group=c(1,1,1,2,2,2,3,3,3), math=c(2,3,2,4,3,5,3,2,4),
+  geo=c(1,2,1,3,3,4,2,2,3))
> dfInd
  group math geo
1     1    2   1
2     1    3   2
3     1    2   1
4     2    4   3
5     2    3   3
6     2    5   4
7     3    3   2
8     3    2   2
9     3    4   3
> dfGr <- data.frame(group=c(1,2,3), size=c(21,40,32))
> dfGr
  group size
1     1   21
2     2   40
3     3   32
```

Um eine Mehrebenenanalyse zu ermöglichen, muss die Tabelle mit den Mikroebenendaten um die Makrovariable `size` erweitert werden (siehe z.B. Bliese 2013), wobei die in beiden Datentabellen vorhandene Variable `group` für eine korrekte Zuordnung sorgen soll. Für das Zusammenführen der Daten eignet sich in **R** die Funktion **merge()**. Man übergibt die beiden Datenquellen als erstes bzw. zweites Argument und definiert anschließend die Zuordnung, was im Beispiel über eine einzige, in beiden Tabellen vorhandene und identisch benannte Variable geschehen kann:

```
> dfIndAug <- merge(dfInd, dfGr, by="group")
> dfIndAug
  group math geo size
1     1    2   1   21
2     1    3   2   21
3     1    2   1   21
4     2    4   3   40
5     2    3   3   40
6     2    5   4   40
7     3    3   2   32
8     3    2   2   32
9     3    4   3   32
```

7.7 Daten aggregieren

Oft sind für die Fälle in einer Datentabelle Gruppierungsvariablen vorhanden, und es wird eine neue Datentabelle benötigt, die für jede Ausprägung einer Gruppierungsvariablen oder für jede Ausprägungskombination von mehreren Gruppierungsvariablen genau einen Fall enthält, wobei die Werte dieses Falles bei den nicht zur Gruppierung verwendeten Variablen durch Aggregieren über die zugehörige Gruppe entstehen. Im folgenden Beispiel, das auch schon in Abschnitt 7.6 verwendet wurde, liegen Leistungsmessungen in den Fächern Mathematik und Geographie für Schüler in drei Gruppen (z.B. Klassen) vor:

```
> dfInd <- data.frame(group=c(1,1,1,2,2,2,3,3,3), math=c(2,3,2,4,3,5,3,2,4),
+  geo=c(1,2,1,3,3,4,2,2,3))
> dfInd
```

```
  group math geo
1     1    2   1
2     1    3   2
3     1    2   1
4     2    4   3
5     2    3   3
6     2    5   4
7     3    3   2
8     3    2   2
9     3    4   3
```

Mit der Funktion **aggregate()** lässt sich daraus eine Datentabelle mit den *Klassen* als Fällen erstellen, wobei die Ausprägungen bei den Variablen `math` und `geo` durch Mittelwertsbildung in den Klassen entstehen:

```
> dfGr <- aggregate(dfInd[,2:3], list(dfInd$group), mean, na.rm=TRUE)
```

Im ersten Argument werden aus `dfInd` per Indexzugriff die Aggregierungsvariablen gewählt. Das zweite Argument legt die Liste mit den Gruppierungsvariablen fest, wobei das Beispiel mit *einer* Variablen auskommt. Im dritten Argument wird die Aggregierungsfunktion **mean** gewählt. Mit dem optionalen Argument **na.rm** wird schließlich dafür gesorgt, dass bei Gruppen mit fehlenden Werten *nicht* das Ergebnis **NA** resultiert, sondern das arithmetische Mittel der *vorhandenen* Werte. Im Beispiel resultiert die Datentabelle:

```
> dfGr
  Group.1     math      geo
1       1 2.333333 1.333333
2       2 4.000000 3.333333
3       3 3.000000 2.333333
```

Um diese Daten auf der Gruppen- bzw. Makroebene (z.B. für eine Mehrebenenanalyse) wieder mit den Daten auf der Mikroebene (in der Tabelle `dfInd`) zusammen zu führen, ist die in Abschnitt 7.6 vorgestellte Funktion **merge()** zu verwenden.

7.8 Sekundärstichproben ziehen

Mit der Funktion **sample()** kann man aus einem Vektor mit einer Primärstichprobe durch Ziehen mit Zurücklegen eine Sekundärstichprobe gleicher Größe gewinnen, z.B.:

```
> prim <- 1:9
> sec <- sample(prim,size=length(prim),replace=TRUE)
[1] 5 9 6 5 2 5 3 2 9
```

Um für einen numerischen Vektor eine zufällige **Permutation** zu gewinnen, zieht man eine Stichprobe vom Originalumfang mit dem voreingestellten Wert **FALSE** für das Argument **replace**.

8 Statistische Datenanalyse mit R

In diesem Abschnitt wird keinesfalls versucht, den enormen **R**-Funktionsumfang zur statistischen Datenanalyse zu beschreiben. Das (leider bei weitem noch nicht erreichte) Ziel besteht in einer Sammlung nützlicher Werkzeuge für Personen, die ihre statistischen Analysen überwiegend mit SPSS Statistics erledigen wollen. In Abschnitt 8.1 werden einfache Funktionen zur univariaten Verteilungsbeschreibung vorgestellt, die sich potentiell zur Verwendung in **R**-Skripten eignen. Abschnitt 8.2 behandelt die etwas gewöhnungsbedürftige **R**-Syntax zur Modellformulierung, die in zahlreichen Auswertungsfunktionen Verwendung findet. Im Abschnitt 8.3 werden exemplarisch einige R-Funktionen beschrieben, die Lücken im Statistikangebot von SPSS schließen. Sie sind über das Formulieren von **R**-Anweisungen zu nutzen, während man bei den in Abschnitt 3 vorgestellten SPSS-Erweiterungskommandos von **R** profitiert, ohne seine Syntax beherrschen zu müssen.

8.1 Einfache univariate Verteilungsbeschreibung

In diesem Abschnitt werden einfache **R**-Funktionen zur Beschreibung von univariaten Verteilungen vorgestellt, die sich potentiell zur Verwendung in eigenen **R**-Skripten eignen. Es geht *nicht* darum, die in **R** zahlreich vorhandenen Schätz- und Testmethoden für univariate Verteilungen vorzustellen.

8.1.1 Univariate Verteilungsbeschreibung für metrische Variablen

8.1.1.1 Kompakte Verteilungsbeschreibung

Von der Funktion **summary()** erhält man für die Stichprobe in einem Vektor das Minimum, das Maximum, das 1., 2. und 3. Quartil sowie den Mittelwert. Im folgenden Beispiel werden die se Statistiken für eine per **rnorm()** (siehe Abschnitt 7.4.1.1) generierte Zufallsstichprobe aus einer normalverteilten Population ermittelt:

```
> summary(rnorm(100, 3, 2))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -2.649   2.126   3.357   3.336   4.515   8.401
```

8.1.1.2 Statistische Funktionen für numerische Vektoren

Statistische Funktionen für numerische Vektoren:

- **min()**, **max()**
  Die Funktionen **min()** bzw. **max()** liefert das kleinste bzw. größte Element, z.B.:
  ```
  > min(c(1,2,3,5))
  [1] 1
  > max(c(1,2,3,5))
  [1] 5
  ```
- **sum()**
  Liefert die Summe der Elemente, z.B.:
  ```
  > sum(c(1,2,3,5))
  [1] 11
  ```
- **mean()**
  Liefert das arithmetische Mittel der Elemente, z.B.:
  ```
  > mean(c(1,2,3,5))
  [1] 2.75
  ```

- **median()**
  Liefert den Median der Elemente, z.B.:
  ```
  > median(c(1,2,3,5))
  [1] 2.5
  ```
- **var()** und **sd()**
  Diese Funktionen liefern die Varianz bzw. Standardabweichung einer Stichprobe, z.B.:
  ```
  > var(c(1,2,3,5,4,5,2,3,1,4,5,7,8,2,3,4,2,3,8))
  [1] 4.508772
  > sd(c(1,2,3,5,4,5,2,3,1,4,5,7,8,2,3,4,2,3,8))
  [1] 2.123387
  ```
- **quantile()**
  Liefert per Voreinstellung die Quartile, z.B.:
  ```
  > quantile(c(1,2,3,5,4,5,2,3,1,4,5,7,8,2,3,4,2,3,8))
    0%  25%  50%  75% 100%
     1    2    3    5    8
  ```
  Über einen Parametervektor mit Wahrscheinlichkeiten sind auch andere Quantile verfügbar, z.B.:
  ```
  > quantile(c(1,2,3,5,4,5,2,3,1,4,5,7,8,2,3,4,2,3,8), c(1/3, 2/3))
  33.33333% 66.66667%
          3         4
  ```
  Als Rückgabe erhält man einen Vektor mit benannten Elementen (vgl. Abschnitt 5.3.4.2.6).
- **IQR()**
  Liefert den Interquartilsabstand (Abstand vom 1. bis zum 3. Quartil), z.B.:
  ```
  > IQR(c(1,2,3,5,4,5,2,3,1,4,5,7,8,2,3,4,2,3,8))
  [1] 3
  ```

Enthält ein Vektor einen fehlenden Wert (**NA** oder **NaN**) liefern die genannten Funktionen das Ergebnis **NA** oder **NaN**, z.B.:

```
> a <- c(1, NA, 3)
> mean(a)
[1] NA
```

Soll stattdessen aus den vorhandenen Argumenten ein Ergebnis ermittelt werden, ist das Argument **na.rm** auf den Wert **TRUE** zu setzen, z.B.:

```
> mean(a, na.rm=TRUE)
[1] 2
```

8.1.2 Absolute und relative Häufigkeiten für kategoriale Variablen ausgeben

Die Funktion **table()** liefert für einen Vektor oder Faktor die absoluten Häufigkeiten der Kategorien, z.B.:

```
> daten <- c(1,1,1,2,2,2,2,3,3,3,3,4,4,5,5,5,5)
> (tabelle <- table(daten))
daten
1 2 3 4 5
3 4 4 2 4
```

Zu den relativen Häufigkeiten verhilft die Funktion **prop.table()**, die eine Tabelle mit absoluten Häufigkeiten als Argument benötigt, z.B.:

```
> prop.table(tabelle)
daten
        1         2         3         4         5 
0.1764706 0.2352941 0.2352941 0.1176471 0.2352941 
```

8.2 Modellformulierung

Ein univariates Modell (z.B. für die Funktionen **lm()**, **glm()** oder **rlm()**) verwendet folgende Syntax:

Kriterium ~ Design

Das zu modellierende Kriterium kann sein:

- eine Variable
- ein Funktionsausdruck, z.B.

  ```
  log(y) ~ x
  ```

Das Design enthält mindestens einen Term. Treten mehrere Terme auf, sind diese durch den **+** - Operator zu trennen, z.B.:

```
y ~ x1 + x2
```

Erlaubte Terme:

- Variablen (z.B. Vektoren oder Faktoren aus einer Datentabelle)
- Die Zahl **1** steht für den konstanten Term eines linearen Modells und muss nicht angegeben werden, weil der konstante Term implizit enthalten ist.
- Wechselwirkungsterme bestehend aus mindestens zwei per **:** - Operator getrennten Variablen, z.B.:

  ```
  y ~ a + b + a:b
  ```

Formuliert man einen Wechselwirkungsterm per ***** - Operator, sind alle Terme niedrigerer Ordnung einbezogen, z.B.:

Kurzschreibweise	enthaltene Terme
a*b	a, b, a:b
a*b*c	a, b, c, a:b, a:c, b:c, a:b:c

Wenn ein Faktor `b` in einem Faktor `a` **geschachtelt** ist, wenn also jede Kategorie von Faktor `b` unter genau einer Kategorie von Faktor `a` auftritt, wird das Design folgendermaßen notiert:

```
y ~ a/b
```

Über den Minus-Operator kann ein Term aus dem Design entfernt werden, was gelegentlich mit dem implizit enthaltenen konstanten Term eines linearen Modells (symbolisiert durch die Zahl **1**) geschieht. Im folgenden Beispiel mit den numerischen Vektoren `x` und `y` resultiert das Modell einer bivariaten linearen Regression mit einer durch den Ursprung des Koordinatensystems verlaufenden Regressionsgeraden:

```
y ~ x - 1
```

Im Design kann an Stelle einer Variablen auch ein Funktionsausdruck stehen, z.B.:

```
y ~ log(x)
```

Werden im Funktionsausdruck Operatoren benötigt, die in der Modellformulierung eine alternative Bedeutung haben, muss per **I()** - Funktion die arithmetische Bedeutung der Operatoren wiederhergestellt werden, z.B.:

```
lm(y ~ I(x1+x2))
```

Weitere, weniger oft benötigte Details zur Modellformulierung finden sich z.B. bei Venables et al. (2014, Abschnitt 11.1) und Wollschläger (2010, S. 163ff).

8.3 Lücken im SPSS-Statistikangebot füllen

In diesem Abschnitt werden weitere statistische Methoden behandelt, die in SPSS fehlen und die mit **R** leicht zu realisieren sind. Einige Ergänzungen der SPSS-Funktionalität konnten schon in Abschnitt 3 vorgestellt werden, weil sie über Erweiterungsbundles ohne **R**-Kenntnisse zu nutzen sind. In Abschnitt 4 wurde erläutert, wie man **R**-Programme im SPSS-Syntaxfenster erstellt und darin auf SPSS-Variablen zugreift. In Abschnitt 5 wurde **R** als Programmierumgebung vorgestellt, so dass wir nun die Angebote im **R**-Universum nahezu uneingeschränkt nutzen können.

8.3.1 Fleiss-Kappa

In einem Beispiel mit fiktiven Daten haben drei Beurteiler zehn Objekte jeweils in eine von fünf Kategorien eingeordnet. Wir erstellen zunächst für jeden Rater einen Vektor mit seinen Urteilen und koppeln diese Vektoren mit der Funktion **cbind()** (vgl. Abschnitt 5.3.4.4.2) zu einer Matrix:

```
> rater1 <- c(1,4,5,3,5,4,3,5,2,3)
> rater2 <- c(1,3,5,3,3,5,2,5,3,3)
> rater3 <- c(4,3,5,3,4,5,3,3,2,3)
> rater <- cbind(rater1,rater2,rater3)
> rater
      rater1 rater2 rater3
 [1,]      1      1      4
 [2,]      4      3      3
 [3,]      5      5      5
 [4,]      3      3      3
 [5,]      5      3      4
 [6,]      4      5      5
 [7,]      3      2      3
 [8,]      5      5      3
 [9,]      2      3      2
[10,]      3      3      3
```

In Abschnitt 5.2.2 haben wir das **R**-Paket **irr** (***I****nter*-***R****ater*-***R****eliability*) installiert. Nun soll es dazu verwendet werden, mit dem **Fleiss-Kappa** ein Maß für die Übereinstimmung von *k* (> 2) Beurteilern bei der Einschätzung eines kategorialen Merkmals zu bestimmen. Wir laden das Paket mit einem Aufruf der Funktion **library()**:

```
> library(irr)
```

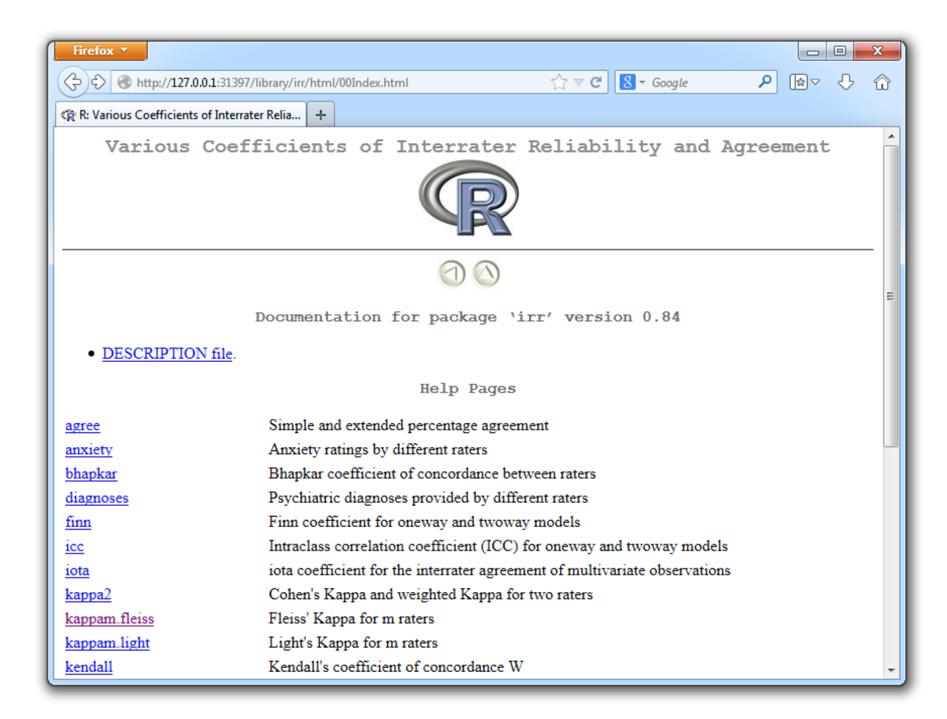
Um die benötigte Funktion aus dem Paket **irr** kennen zu lernen, starten wir mit der Anweisung

```
> help.start()
```

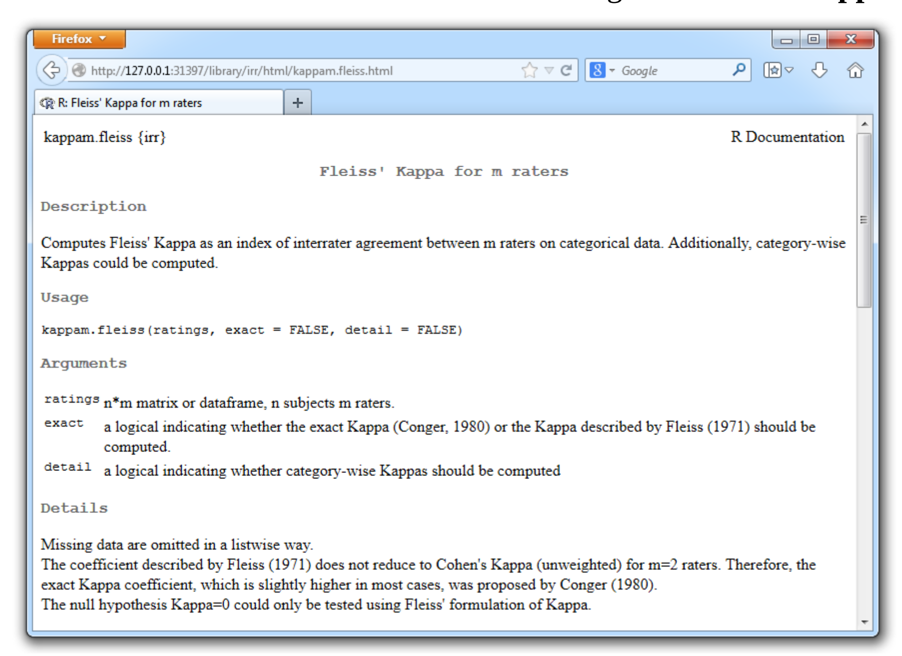
oder mit dem RGui-Menübefehl

Hilfe > HTML-Hilfe

die HTML-Hilfe zu **R**, klicken zunächst auf den Link **Packages** und in der nun erscheinenden Paketliste auf den Paketnamen **irr**. In der angezeigten Paketbeschreibung ist die gesuchte Funktion zum Fleiss-Kappa leicht zu finden:



Durch einen Klick auf ihren Namen erhält man eine Beschreibung der Funktion **kappam.fleiss()**,



die bei geladenem Paket auch über das Kommando

```
> ?kappam.fleiss
```

abrufbar ist.

Wie die Beispiele in der Beschreibung zeigen, genügt zum Aufruf der Funktion eine Angabe der Daten im Argument **ratings**, wobei eine Matrix oder eine Datentabelle akzeptiert werden. Wir können für unsere Daten also den folgenden Aufruf verwenden:

```
> kappam.fleiss(rater)
Fleiss' Kappa for m Raters

 Subjects = 10
   Raters = 3
    Kappa = 0.295

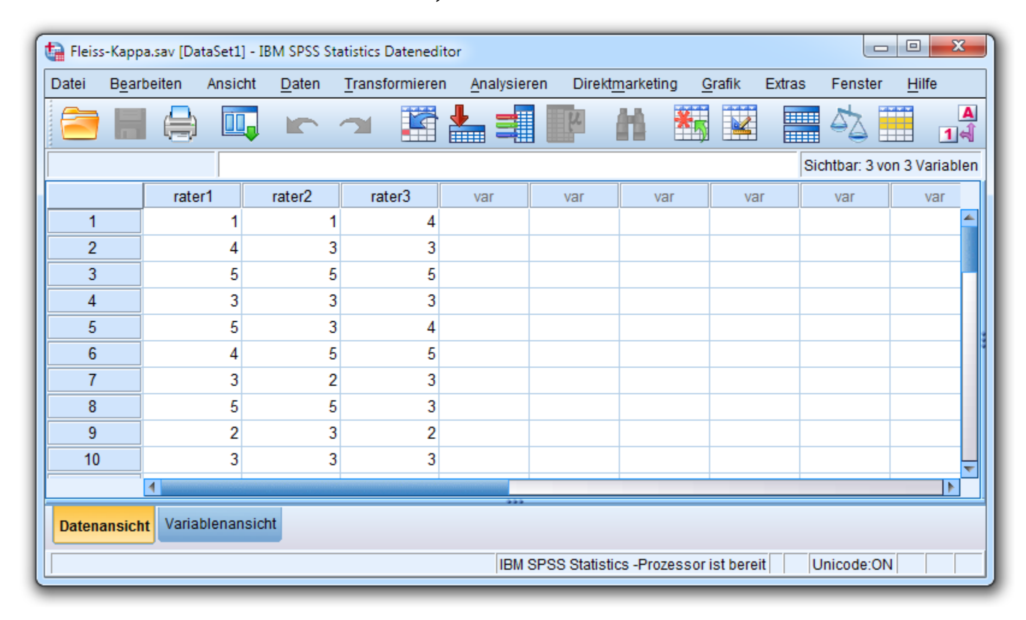
        z = 2.79
  p-value = 0.00523
```

Für das Fleiss-Kappa resultiert der Schätzwert 0,295, und die Nullhypothese

$$H_0\text{: Kappa} = 0$$

wird durch den p-Wert von 0,005 klar verworfen. Allerdings ist eine signifikant von 0 verschiedene Übereinstimmung noch keine Garantie für die diagnostische Tauglichkeit der Urteilsleistung.

Liegen die Daten in der SPSS-Arbeitsdatei vor,



	rater1	rater2	rater3
1	1	1	4
2	4	3	3
3	5	5	5
4	3	3	3
5	5	3	4
6	4	5	5
7	3	2	3
8	5	5	3
9	2	3	2
10	3	3	3

gelingt die Fleiss-Kappa - Berechnung z.B. mit der folgenden Syntax:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
library(irr)
kappam.fleiss(data)
END PROGRAM.
```

8.3.2 Gerichtete t-Tests mit einseitigem Vertrauensintervall

Bei t-Tests liefert SPSS Statistics grundsätzlich nur den p-Wert zum *zweiseitigen* Test und dazu passend das zweiseitige Konfidenzintervall. Wenn ein *gerichtetes* Testproblem vorliegt, kann der benötigte einseitige p-Wert leicht berechnen werden (durch Halbieren des zweiseitigen p-Werts). Das zugehörige einseitige Vertrauensintervall zu bestimmen, ist etwas umständlich. Diese Mühe kann man sich ersparen durch Verwendung der **R**-Funktion **t.test()**, die über das Argument **alternative** mit den Werten

- **"two.sided"** (Voreinstellung),
- **"greater"** oder
- **"less"**

eine Testausrichtung entgegennimmt und bei einseitiger Testung auch ein passendes einseitiges Vertrauensintervall berechnet. Wir rechnen als Beispiel einen Einstichproben - t-Test zum Hypothesenpaar

$$H_0: \mu \leq 0 \text{ versus } H_1: \mu > 0$$

und verwenden dazu eine Zufallsstichprobe (*N* = 50) aus einer normalverteilten Population mit dem Mittelwert 0,2 und der Varianz 1:

```
> sam <- rnorm(50, 0.2, 1)
> t.test(sam)
> t.test(sam, alternative="greater")
```

Beim ungerichteten t-Test als Folge des ersten **t.test()** - Aufrufs resultiert ein p-Level von 0,063, und die Nullhypothese muss beibehalten werden. Das zweiseitige Konfidenzintervall (zum voreingestellten Niveau von 0,95) enthält dementsprechend den Wert Null:

```
        One Sample t-test

data:  sam
t = 1.9022, df = 49, p-value = 0.06303
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.01514014  0.55151178
sample estimates:
mean of x
0.2681858
```

Beim gerichteten t-Test als Folge des zweiten **t.test()** - Aufrufs resultiert ein p-Level kleiner als 0,05, so dass die einseitige Nullhypothese verworfen werden kann. Das einseitige Konfidenzintervall liegt dementsprechend komplett rechts von der 0:

```
        One Sample t-test

data:  sam
t = 1.9022, df = 49, p-value = 0.03152
alternative hypothesis: true mean is greater than 0
95 percent confidence interval:
 0.03181230        Inf
sample estimates:
mean of x
0.2681858
```

9 Grafik-Optionen in R

Das traditionelle Grafiksystem in **R** basiert auf dem stets installierten und auch automatisch geladenen Paket **graphics**. Daneben bietet **R** als alternatives Grafiksystem die so genannte *Gittergrafik* (engl. *grid graphics*), auf der folgende Optionen zur Grafikproduktion basieren:

- **Trellis-Grafiken**
  Das zur Grundinstallation von **R** gehörige Paket **lattice** enthält die so genannten *Trellis-Grafiken* (siehe z.B. Murrell 2011).
- **Grammar of Graphics**
  Das zusätzlich zu installierende, von Wickham (2009) erstellte Paket **ggplot2** basiert auf der von Wilkinson (2005) entwickelten *Grammar of Graphics* (daher der Namensanfang *gg*). Dieselbe Grundlogik ist übrigens auch in der von SPSS unterstützten *Graphics Production Language* (GPL) realisiert.

Im Manuskript werden das traditionelle Grafiksystem sowie die Grafikproduktion mit dem Paket **ggplot2** behandelt (siehe Abschnitte 9.2 bzw. 9.3). Im Wettbewerb der **R**-Grafik-Optionen zeichnet sich derzeit eine Tendenz zur Bevorzugung des Pakets **ggplot2** ab (siehe z.B. Field 2012, S. 121). Robert Muenchen (2011, S. 444f) urteilt:

> The point is that **ggplot2** gives you the broadest range of graphical options that R offers.

Wer dieser Empfehlung folgen möchte, kann im Manuskript den Abschnitt 9.2 über die traditionelle Grafikproduktion auslassen. Im allgemein relevanten Abschnitt 9.1 wird allerdings in einigen Beispielen das ohne jedes Installieren und/oder Laden von Paketen verfügbare traditionelle Grafiksystem eingesetzt.

9.1 Ausgabeformate und -geräte

Die Ausgaben der Grafikfunktionen werden zu einem Ausgabegerät kanalisiert, das eine Bildschirm-, Datei- oder Druckausgabe produziert.

9.1.1 Verfügbare Ausgabegeräte

Zum Öffnen eines Ausgabegeräts von bestimmtem Typ ist jeweils die zuständige Funktion aufzurufen. Über die Namen dieser Funktionen und damit über die unterstützten Ausgabeformate informiert die **R**-Hilfe nach der Anweisung

```
> ?device
```

Bei **R** 2.15.3 erscheint unter Windows die folgende Geräteliste:

- **windows** The graphics device for Windows (on screen, to printer and to Windows metafile)
- **pdf** Write PDF graphics commands to a file
- **postscript** Writes PostScript graphics commands to a file
- **xfig** Device for XFIG graphics file format
- **bitmap** bitmap pseudo-device via Ghostscript (if available)
- **pictex** Writes TeX/PicTeX graphics commands to a file (of historical interest only)
- **cairo_pdf**, **cairo_ps** PDF and PostScript devices based on cairo graphics
- **svg** SVG device based on cairo graphics
- **png** PNG bitmap device
- **jpeg** JPEG bitmap device
- **bmp** BMP bitmap device
- **tiff** TIFF bitmap device

Während das erste Gerät in dieser Liste zur Ausgabe in ein Grafikfenster (siehe Abschnitt 9.1.2) taugt, sind die anderen Geräte jeweils mit einer Datei verbunden.

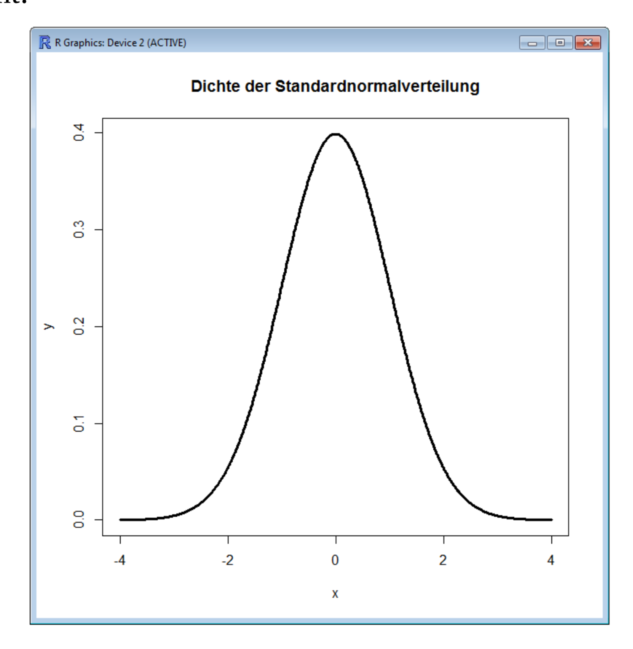
Das in der obigen Liste fehlende Gerät zur Direktausgabe in eine Datei vom Typ *Enhanced Windows Metafile* (*EMF*) wird (zumindest in der Windows-Version von **R**) durch die Funktion **win.metafile()** bereitgestellt.

9.1.2 Grafikfenster

Bei Verwendung der RGui-Bedienoberfläche ist das voreingestellte Ausgabegerät ein Grafikfenster, das beim ersten Aufruf einer Grafikfunktion automatisch initialisiert wird. Als Beispiel erstellen wir mit dem traditionellen Grafiksystem ein Liniendiagramm mit der Standardnormalverteilungsdichte.

```
> x <- seq(-4,4,0.01); y <- dnorm(x, mean = 0, sd = 1)
> plot(x,y, type="l", main="Dichte der Standardnormalverteilung", lwd=3)
```

Per **seq()**-Funktion (siehe Abschnitt 5.3.4.2.1) entsteht ein Vektor mit X-Koordinaten. Von der **dnorm()** - Funktion erhalten wir einen gleichlangen Y-Vektor mit den Funktionswerten der Standardnormalverteilungsdichte zu den X-Werten. Per **plot()** - Funktion entsteht ein Liniendiagramm mit der Dichte, das in einem Grafikfenster erscheint:



Dieses Ausgabegerät hat den Typ **windows** und kann mit der gleichnamigen Funktion auch explizit erstellt werden, z.B.

```
> windows(5, 5)
```

Dieses Vorgehen bietet die Möglichkeit, eine alternative Breite und Höhe anstelle der voreingestellten Werte von jeweils 7 Zoll (engl.: *Inch*) festzulegen (= 7 · 2,54 cm = 17,78 cm).

Durch jeden Aufruf der Funktion **windows()** wird ein neues Grafikfenster erstellt und zum aktiven Ausgabegerät ernannt (vgl. Abschnitt 9.1.4 zur Verwaltung der Ausgabegeräte).

Jedes Ausgabegerät hat neben seinem Typ auch eine Nummer, die bei einem Grafikfenster in der Titelzeile erscheint (siehe Beispiel).

Ist im RGui ein Grafikfenster aktiv, kann sein Inhalt ...

- über den Menübefehl **Datei > Speichern als** in diversen Dateiformaten gespeichert werden (**Metafile**, **Postscript**, **PDF**, **Png**, **Bmp**, **TIFF** und **Jpeg**),
- über den Menübefehl **Datei > Kopieren in Zwischenablage als Metafile** oder **Bitmap** in die Windows-Zwischenablage kopiert werden,
- über den Menübefehl **Datei > Drucken** auf einem Drucker ausgegeben werden.

Über das Kontextmenü zum Grafikfenster sind (abgesehen von einer Beschränkung auf die Dateiformate Metafile und Bitmap) dieselben Aktionen möglich.

Soll ein Diagramm in ein Word-Dokument integriert werden, ist aufgrund der Export/Import - Schnittmenge von **R** und Word ein Bitmap-Format zu verwenden. Weil bei der Diagrammerstellung via Grafikfenster die Bitmap-Auflösung nicht beeinflusst werden kann, scheidet für die Übergabe von **R** an Word der bequeme Weg über die Windows-Zwischenablage aus. Um eine gute Qualität zu erzielen, sollte über ein alternatives Ausgabegerät eine Datei erstellt und diese in das Textdokument eingefügt werden (siehe Abschnitt 9.1.3).

Bei aktivem Grafikfenster lässt sich über den Menübefehl

History > Aufzeichnen

die Aufzeichnung der erzeugten Grafiken ein- bzw. ausschalten. Zwischen aufgezeichneten Diagrammen kann man über die Tasten Bild↑ bzw. Bild↓ wechseln.

Um ein Grafikfenster mit Aufzeichnung per Syntax zu öffnen, wählt man den Funktionsaufruf:

```
> windows(record=TRUE)
```

9.1.3 Ausgabe in eine Datei

Unter Windows stehen zwei Möglichkeiten zur Verfügung, um ein Diagramm in eine Datei zu schreiben:

- Man erstellt ein Grafikfenster durch die implizite oder explizite Verwendung der Funktion **windows()** und speichert seinen Inhalt in eine Datei, was über den Menübefehl

  Datei > Speichern als

  in den Formaten **Metafile**, **Postscript**, **PDF**, **Png**, **Bmp**, **TIFF** und **Jpeg** gelingt. Mit Ausnahme der Dateiformate Postscript und PDF erhält man ein Bitmap-Ergebnis mit einer bescheidenen Auflösung, die für Publikationszwecke nicht ausreicht.
- Man öffnet ein Ausgabegerät, das mit einer Datei verbunden ist, durch Aufruf der Format-spezifischen Funktion (z.B. **svg()**, **png()**, **win.metafile()**). Bei einem Bitmap-Format sollten die Größe und die Auflösung korrekt gewählt werden, um eine gute Qualität zu erzielen, was im anschließenden Beispiel demonstriert wird.

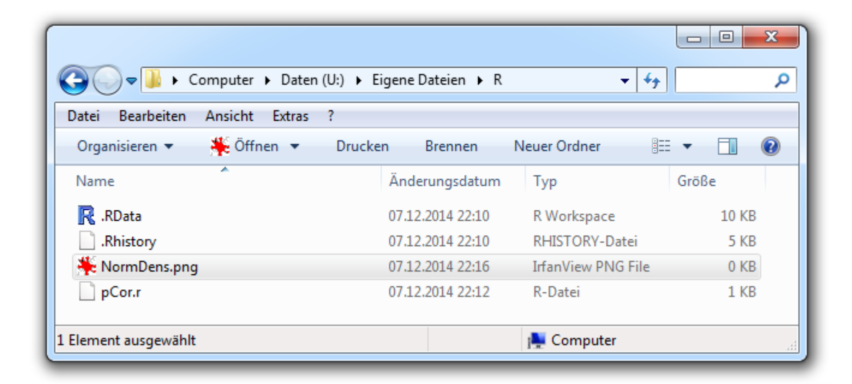
Im folgenden Beispiel entsteht eine PNG-Datei mit einer Breite und Höhe von 10 cm bei einer Auflösung von 600 dpi:

```
> png("NormDens.png", 10, 10, units="cm", res=600)
```

Nach dem Öffnen des Ausgabegeräts führt man die Grafikfunktionsaufrufe durch, z.B.:

```
> plot(x, y, type="l", main="Dichte der Standardnormalverteilung", lwd=3)
```

Daraufhin wird die Ausgabedatei (mit der Größe von 0 Bytes) angelegt und ist durch das RGui blockiert:



Nach Fertigstellung des Diagramms schließt man das Ausgabegerät, z.B.:

```
> dev.off()
windows
      2
```

Daraufhin erfolgt die Ausgabe, und die Datei wird zur Verwendung durch andere Programme freigegeben. Außerdem erfährt man Typ und Nummer des nunmehr aktiven Ausgabegeräts.

9.1.4 Ausgabegeräte verwalten

Über die Funktion **dev.list()** erhält man eine Liste der geöffneten Ausgabegeräte, z.B.:

```
> dev.list()
         windows png:NormDens.png
               2                3
```

Sind mehrere Ausgabegeräte offen, ist eines als **ACTIVE** ausgezeichnet und das Ziel für die Ausgabe von Grafikfunktionen. Man erfährt seinen Typ und seine Nummer über die Funktion **dev.cur()**, z.B.:

```
> dev.cur()
png:NormDens.png
               3
```

Um ein anderes Ausgabegerät zu aktivieren, ruft man die Funktion **dev.set()** mit der Gerätenummer auf, z.B.:

```
> dev.set(2)
windows
      2
```

Eine Besonderheit der Geräte **postscript** und **pdf** besteht darin, dass mehrere Aufrufe von High-Level - Grafikfunktionen zu mehrseitigen Dokumenten führen, während bei sonstigen Ausgabegeräten ein neues Diagramm das bisherige ersetzt.

Auf das aktive Grafikfenster hat der Funktionsaufruf **dev.off()** denselben Effekt wie ein Klick auf das Schließkreuz in der Titelzeile.

Um *alle* Ausgabegeräte zu schließen, verwendet man die Anweisung:

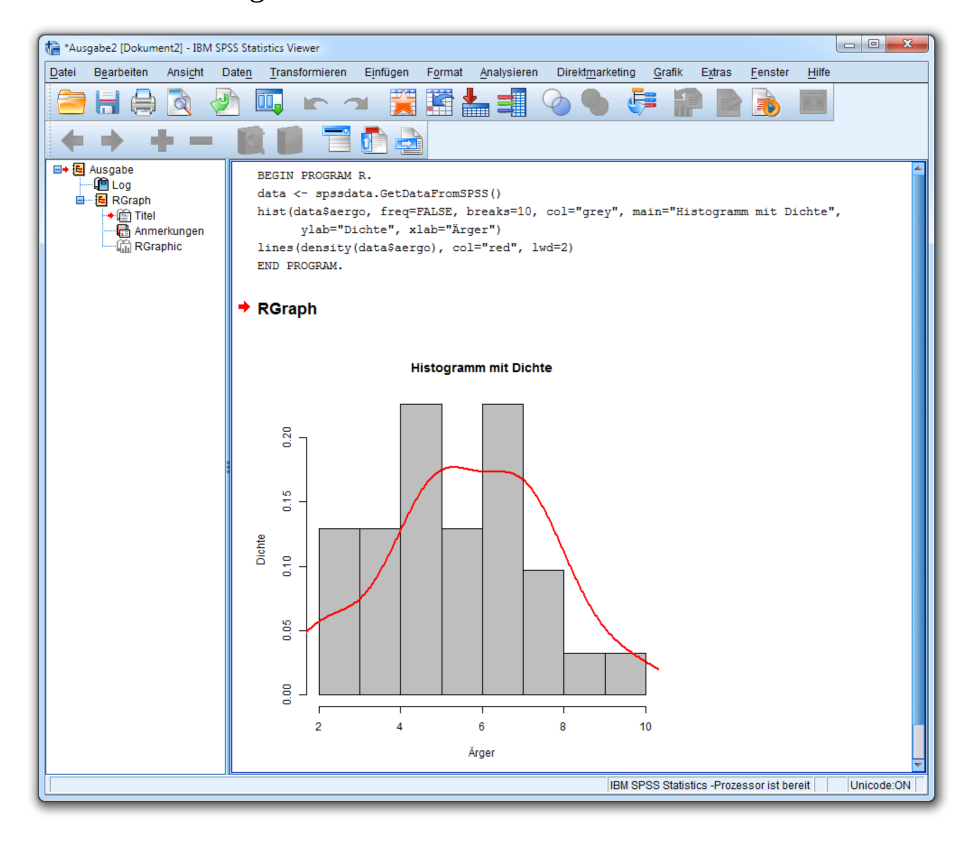
```
> graphics.off()
```

9.1.5 R-Diagramme im SPSS-Ausgabefenster

Es ist selbstverständlich möglich, **R**-Diagramme per SPSS-Syntaxfenster zu erstellen, und im Abschnitt 9.2 werden wir von dieser Möglichkeit mehrfach Gebrauch machen. Im folgenden Beispiel wird mit Funktionen aus dem traditionellen Grafiksystem von **R** ein Histogramm mit Dichteschätzung zu einer Variablen der SPSS-Arbeitsdatei erstellt:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
hist(data$aergo, freq=FALSE, breaks=10, col="grey", main="Histogramm mit Dichte",
     ylab="Dichte", xlab="Ärger")
lines(density(data$aergo), col="red", lwd=2)
END PROGRAM.
```

Das Ergebnis landet im SPSS-Ausgabefenster



und kann von dort via Windows-Zwischenablage in andere Anwendungen (z.B. Word) weiterbefördert werden.

Die Qualität der **R**-Diagramme im SPSS-Ausgabefenster reicht für den Forschungsalltag, aber in der Regel nicht für Publikationszwecke. Bei höheren Qualitätsansprüchen empfiehlt sich der Weg über eine Datei als Grafikausgabegerät von **R** (siehe Abschnitt 9.1.3). Für diesen Weg kann man das RGui benutzen oder das SPSS-Syntaxfenster, z.B.:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
png("u:/eigene dateien/r/NormDens.png", 10, 10, units="cm", res=600)
hist(data$aergo, freq=FALSE, breaks=10, col="grey", main="Histogramm mit Dichte",
     ylab="Dichte", xlab="Ärger")
lines(density(data$aergo), col="red", lwd=2)
END PROGRAM.
```

9.2 Das traditionelle Grafiksystem

In diesem Abschnitt wird eine kleine Auswahl der in **R** verfügbaren traditionellen Optionen zur Darstellung von Daten und mathematischen Funktionen vorgestellt.

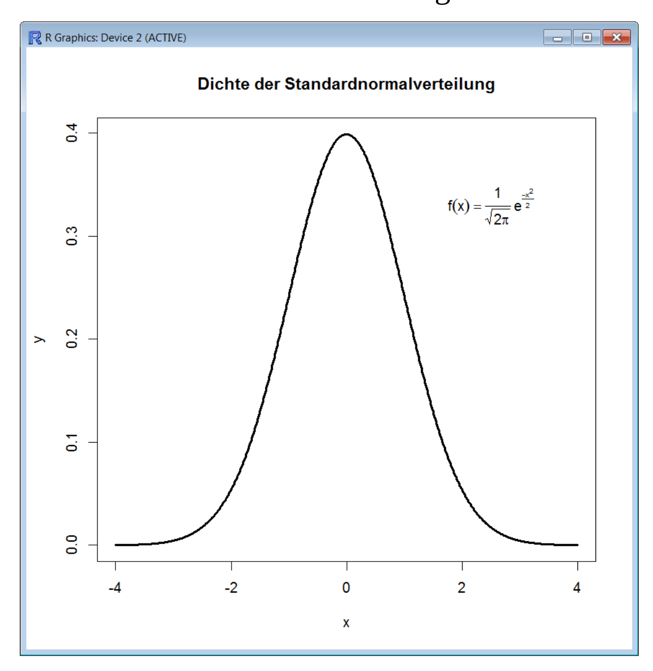
9.2.1 High- und Low-Level - Grafikfunktionen

Die traditionelle **R-Grafik** kennt High-Level - Funktionen, die ein vollständiges Diagramm erstellen (z.B. **plot()**) und Low-Level - Funktionen, die ein Diagramm um ein Element (z.B. eine Beschriftung) erweitern (z.B. **text()**). Das schafft Flexibilität und sorgt dafür, dass sich die traditionelle Grafik gegen starke Konkurrenz im **R**-Universum immer noch behaupten kann.

Mit den folgenden Anweisungen

```
> x <- seq(-4,4,0.01)
> y <- dnorm(x, mean = 0, sd = 1)
> plot(x,y, type="l", main="Dichte der Standardnormalverteilung", lwd=3)
> nd <- expression(f(x) == frac(1,sqrt(2*pi))~e^{frac(-x^2,2)})
> text(2.5, 0.33, nd)
```

wird per **plot()**-Funktion ein Liniendiagramm mit der Standardnormalverteilungsdichte gezeichnet und anschließend per **text()**-Funktion die Definition der Dichte ergänzt:



Mit der **expression()** - Funktion lässt sich mathematische Notation in Diagramme einfügen. Das Beispiel stammt aus Ligges (2007, S. 170).

9.2.2 Grafikparameter und Beschriftungen

R kennt ca. 70 Grafikparameter, die sich für ein Ausgabegerät oder für einen Grafikfunktionsaufruf setzen lassen:

- Sollen Parameter mit Gültigkeit bei allen Grafikfunktionsaufrufen für das aktuelle Ausgabegerät gesetzt werden, ist die Funktion **par()** mit der Syntax

  par(*name*=*wert***,** ...)

  zu verwenden. Ein **par()** - Aufruf ohne Argumente protokolliert die aktuellen Werte der Grafikparameter für das aktive Ausgabegerät.

- Viele Parameter lassen sich auch als Argumente in einem konkreten Grafikfunktionsaufruf (High- oder Low-Level) mit lokaler Gültigkeit verwenden.

Anschließend werden einige Parameter vorgestellt, die entweder für eine generelle Einstellung per **par()** in Frage kommen oder für verschiedene Grafikfunktionen relevant sind. Später folgen noch Parameter, die vor allem für spezielle Grafikfunktionen (z.B. **plot()**) von Interesse sind. Die folgende Liste eignet sich weniger zum Studieren als zum Nachschlagen:

- **col**
  Die Zeichenfarbe kann über einen Farbnamen oder einen RGB-Wert aus drei Hexadezimalzahlen von 00 bis FF (optional ergänzt durch einen Transparenzwert im selben Wertebereich) festgelegt werden, z.B.:
  ```
  > boxplot(x, col="red")
  > boxplot(x, col="#ff0000")    # Rot
  > boxplot(x, col="#ff000080")  # Rot, halb-transparent
  ```
  Über die Funktion **colors()** erhält man einen Vektor mit den 657 verfügbaren Farbnamen.[1]
- **col.axis, col.lab, col.main, col.sub**
  Mit diesen Parametern wird die Farbe für bestimmte Bestandteile eines Diagramms gewählt: Teilstrichbeschriftungen (**col.axis**), Achsenbeschriftungen (**col.lab**), Überschriften erster und zweiter Ordnung (**col.main**, **col.sub**).
- **bg**
  Mit dem Parameter **bg** wird die Hintergrundfarbe für ein Ausgabegerät gesetzt. Voreinstellung für **bg** ist Weiß. Einige Grafikfunktionen (z.B. **points()**) haben ein Argument mit demselben Namen, aber unterschiedlicher Bedeutung.
- **fg**
  Mit dem Parameter **fg** wird die Vordergrundfarbe gesetzt. Geschieht dies in einer Grafikfunktion, sind vor allem Achsen und Rahmen betroffen. Geschieht es in einem **par()** - Aufruf, wird der Parameter **col** auf denselben Wert gesetzt. Voreinstellung für **fg** ist Schwarz.
- **family**
  Mit diesem Parameter wählt man eine Schriftfamilie. Generell verfügbar sind:

Familienname	**Unter Windows abgebildet auf**
mono	TT Courier New
serif	TT Times New Roman
sans	TT Arial

  Über die Funktion **windowsFonts()** lassen sich andere im Windows-Betriebssystem des Rechners installierte Schriftarten einbinden (siehe **R**-Hilfe).
- **font**, **font.axis**, **font.lab**, **font.main**, **font.sub**
  Mit diesem Parameter wählt man eine Schriftauszeichnung:

Wert	**Auszeichnung**
1	normal
2	**fett**
3	*kursiv*
4	***fett-kursiv***
5	Verwendet die Schriftart Symbol

[1] Der folgenden PDF-Datei (abgerufen am 24.12.2014) sind die in **R** verfügbaren Farbnamen zu entnehmen:
http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

Diese Einstellung kann für die gesamte Grafik (**font**) oder für die Teilstrichbeschriftungen (**font.axis**), die Achsenbeschriftungen (**font.lab**), sowie die Überschriften erster und zweiter Ordnung (**font.main**, **font.sub**) vorgenommen werden.

- **ps**
  Dieser Parameter legt die generelle Schriftgröße für ein Ausgabegerät in Point Size - Einheiten fest (eine Einheit = 1/72 Zoll, voreingestellter **ps**-Wert: 12). Wie sich daraus über Skalierungsfaktoren die Größen für spezielle Beschriftungen ableiten lassen, wird anschließend beschrieben.
- **cex, cex.axis, cex.lab, cex.main, cex.sub**
  Durch diesen positiven Skalierungsfaktor kann Größe von Symbolen (**cex**), Teilstrichbeschriftungen (**cex.axis**), Achsenbeschriftungen (**cex.lab**), sowie Überschriften erster und zweiter Ordnung (**cex.main**, **cex.sub**) im Vergleich zum Standard (Wert = 1) reduziert (Wert < 1) oder erhöht werden (Wert > 1).
- **lwd**
  Durch diesen positiven Skalierungsfaktor kann die Linienstärke im Vergleich zum Standard (Wert = 1) reduziert (Wert < 1) oder erhöht werden (Wert > 1), z.B.:
  ```
  > boxplot(x, lwd=2)
  ```

Mit den folgenden Argumenten lassen sich in High-Level - Grafikfunktionen Beschriftungen einfügen:

- **main, sub**
  Titel und Untertitel
- **xlab**, **ylab**
  Beschriftungen für die X- bzw. Y-Achse
  Um die voreingestellten Beschriftungen abzuschalten, weißt man den Argumenten eine leere Zeichenfolge oder den Wert **NA** zu, z.B.:
  ```
  > plot(x, y, type="S", xlab=NA, ylab=NA)
  ```

9.2.3 Die generische Funktion plot()

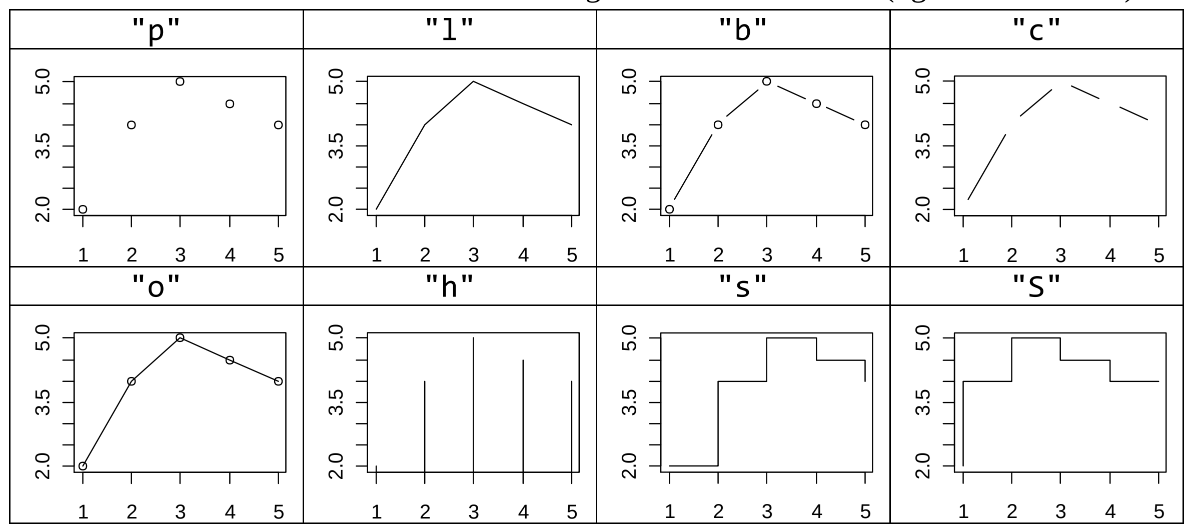
Mit der generischen High-Level - Grafikfunktion **plot()** erstellt man für *k* Koordinatenpaare ((*x*, *y*)-Punkte) ein Liniendiagramm (siehe Abschnitt 9.2.4.1) oder ein Streudiagramm (siehe Abschnitt 9.2.4.2).

9.2.3.1 Argumente

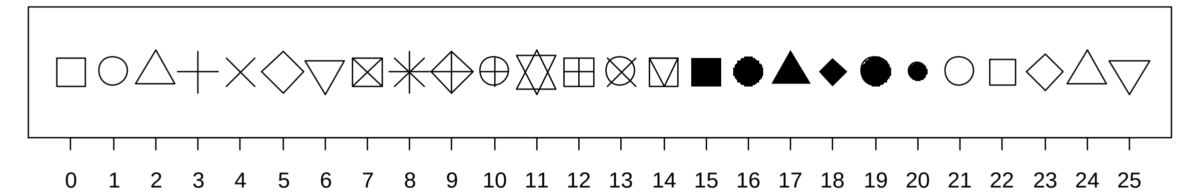
Die Funktion **plot()** kennt u.a. die folgenden Argumente:

- **x**
  k-elementiger Vektor mit den *x*-Koordinaten der Punkte
  Lässt man den **x**-Vektor weg, verwendet **R** die Indexnummern der **y**-Elemente (1, 2, 3, ...).
- **y**
  k-elementiger Vektor mit den *y*-Koordinaten der Punkte
- **type**
  Zur Gestaltung der Punktmarkierungen und Linien stehen folgende **Plot-Typen** zur Verfügung:
  - `"p"` Nur Punkte (= Voreinstellung)
  - `"l"` Nur Linien
  - `"b"` Beides (Punkte und Linien)
  - `"c"` Nur die Linien aus Typ `"b"`
  - `"o"` Beides (Punkte und Linien, „**o**verplotted")
  - `"h"` Senkrechte Linien analog zu einem Histogramm
  - `"s"`, `"S"` Treppenstufen in zwei Varianten
  - `"n"` leeres Diagramm zur Vorbereitung für spätere Low-Level - Ausgaben

Um bei den folgenden Plot-Typ – Demonstrationen Platz zu sparen, wurde als Ausgabegerät ein Grafikfenster mit einer Breite und Höhe von lediglich 3 Zoll verwendet (vgl. Abschnitt 9.1):



- **pch**
  Das Symbol für die Datenpunkte kann aus der folgenden Palette[1]

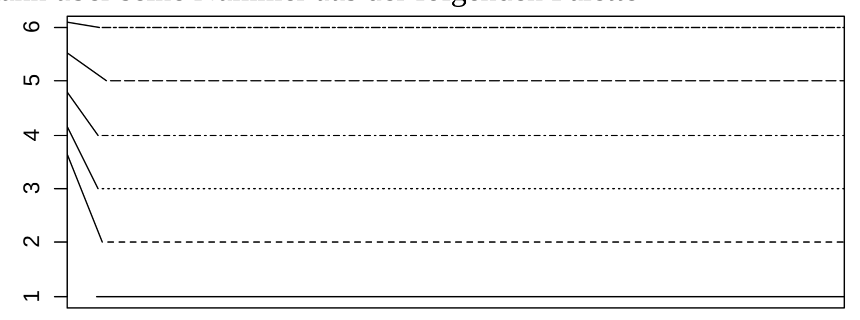


  über seine Nummer gewählt werden, z.B.:

```
> plot(x, y, pch=4)
```

  Statt für alle Datenpunkte dasselbe Symbol zu verwenden, kann man dem Argument **pch** einen Vektor übergeben, der für jeden Datenpunkt ein eigenes Symbol enthält (siehe Abschnitt 9.2.4.2.3).

- **lty**
  Der Linientyp kann über seine Nummer aus der folgenden Palette[2]



[1] Die Abbildung wurde folgendermaßen mit **R** erstellt:

```
> windows(10, 3)
> plot(0:25, rep(1,26), pch=0:25, cex=3, xlab="", ylab="", yaxt="n")
> axis(side = 1, at = 0:25)
```

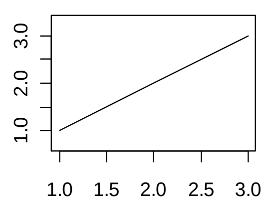
[2] Die Abbildung wurde folgendermaßen mit **R** erstellt:

```
> windows(7,4)
> plot(1:6, 1:6, type="n", xlab="", ylab="", xaxt="n")
> for (i in 1:6) lines(1:6, rep(i,6), lty=i)
```

gewählt werden, z.B.:

```
> plot(x, y, lty=4)
```

- **xlim**, **ylim**
  Begrenzung des Darstellungsbereichs für die X- bzw. Y-Achse, z.B.:
  ```
  > plot(x, y, type="S", xlim=c(-3,3))
  ```
- **xaxt, yaxt, axes**
  Ist die Erstellung individueller Achsen über die Low-Level - Funktion **axis()** geplant (siehe Abschnitt 9.2.3.2), schaltet man die zu ersetzenden Standardvarianten ab:
  - Der Wert **"n"** für das Argument **xaxt** bzw. **yaxt** verhindert die X- bzw. Y-Achse.
  - Der Wert **FALSE** für das Argument **axes** verhindert *beide* Achsen.
- **asp**
  Dieses Argument bestimmt den Quotienten aus der Länge einer Y-Achseneinheit und der Länge einer X-Achseneinheit. Im folgenden Beispiel ist der Abstand zwischen den X-Werten 1 und 2 doppelt so groß wie der Abstand zwischen den Y-Werten 1 und 2:



Neben ihren eigenen Argumenten versteht die Funktion **plot()** auch viele generelle Grafikparameter (siehe Abschnitt 9.2.2), z.B. zur Änderung der Zeichenfarbe mit dem Argument **col**:

```
> plot(x, y, col="red")
```

Statt für alle Datenpunkte dieselbe Farbe zu verwenden, kann man dem Argument **col** einen Vektor übergeben, der für jeden Datenpunkt eine eigene Farbe enthält (siehe Abschnitt 9.2.4.2.3).

9.2.3.2 Ergänzende Low-Level - Grafikfunktionen

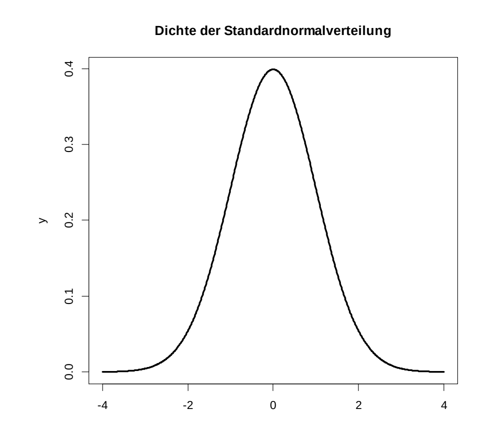
Zur Ergänzung zur **plot()** - Ausgabe kommen u.a. die folgenden Low-Level - Grafikfunktionen in Frage:

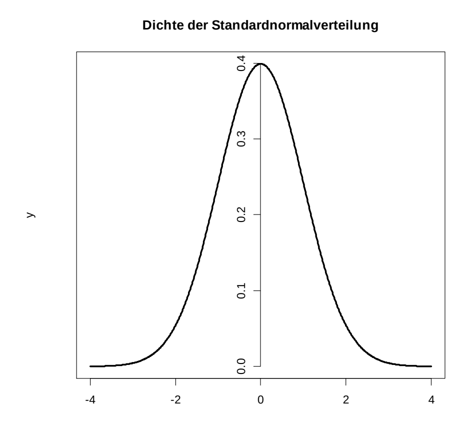
9.2.3.2.1 Achsengestaltung mit der Funktion **axis**()

Um eine spezielle Achsengestaltung zu erreichen, schaltet man die Standardvariante über das **plot()** - Argument **xaxt** bzw. **yaxt** ab (siehe Abschnitt 9.2.3.1) und erstellt mit der **axis()** - Funktion ein individuelles Exemplar, wobei (neben generellen Grafikparametern, siehe Abschnitt 9.2.2) folgenden Argumente verfügbar sind:

- **side**
  Über eine Zahl wird der Erscheinungsort der Achse festgelegt (1 = unten, 2 = links, 3 = oben, 4 = rechts).
- **at**
  Ein numerischer Vektor mit den Positionen für die Teilstriche, z.B.:
  ```
  axis(side = 1, at = 0:25)
  ```
- **tck**
  Über einen Anteil der Zeichenflächengröße legt man die Länge der Teilstriche fest, wobei positive Werte auf der Innenseite und negative Werte auf der Außenseite der Achse erscheinen (Voreinstellung = -0,02). Mit dem Wert 1 erhält man Gitterlinien.

- **labels**
  Mit diesem Zeichenketten-Vektor kann man die Teilstrichbeschriftungen festlegen. Unterlässt man es, kommen die Werte im Vektor **at** zum Einsatz.
- **las**
  Sollen bei einer senkrechten Achse (**side** gleich 2 oder 4) die Teilstrichbeschriftungen parallel zur Achse (Wert = 0) oder orthogonal zur Achse (Wert = 1) geschrieben werden?
- **pos**
  Per Voreinstellung landet eine Achse am Rand des Ausgaberechtecks (siehe linkes Beispiel).





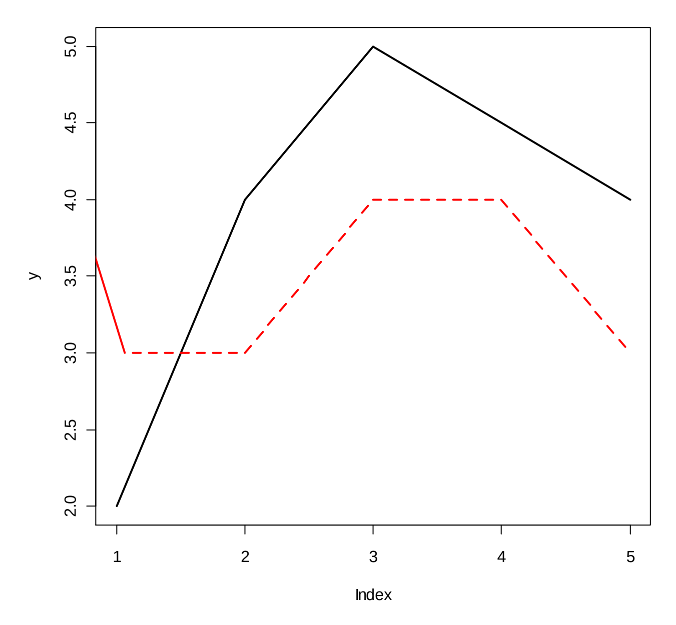
Mit dem Argument **pos** lässt sich der Schnittpunkt mit der orthogonalen Achse festlegen, was im rechten Beispiel für die Y-Achse geschehen ist (**pos**=0).

9.2.3.2.2 Linienzüge ergänzen mit der Funktion **lines**()

Mit der Funktion **lines()** fügt man zusätzliche Linienzüge ein, wobei die Daten (Argumente **x** und **y**) und der Linientyp (Argument **lty**) genauso konfiguriert werden wie bei der **plot()** - Funktion. Als Plot-Typ (Argument **type**) ist bei der **lines()** - Funktion der meist passende Wert "**l**" voreingestellt. Die folgenden Anweisungen

```
> x <- 1:5; y <- c(2, 4, 5, 4.5, 4); y2 <- c(3,3,4,4,3)
> plot(x, y, type="l", lwd=2)
> lines(x,y2, col="red", lty=2, lwd=2)
```

produzieren dieses Diagramm:



Bei einem Streudiagramm (siehe Abschnitt 9.2.4.2) kann die **lines()** - Funktion z.B. dazu dienen, eine lokal optimierte Modellprognose einzuzeichnen:

```
> lines(lowess(daten$gewicht ~ daten$groesse), lty=2, col="blue")
```

9.2.3.2.3 Legenden bilden mit der Funktion **legend**()

Wenn ein Diagramm *mehrere* Linien oder mehrere Datengruppen (Symbole) enthält, ist eine Legende zur Erläuterung der Linien bzw. Gruppen empfehlenswert. Die dafür zuständige Low-Level - Funktion **legend()** kennt u.a. die folgenden generellen Argumente (unabhängig vom Grafiktyp):

- **x**, **y**
  Die Koordinaten der linken, oberen Ecke des Legendenrahmens
  Statt über ein (**x**, **y**)-Zahlenpaar kann man den Ort auch über ein Schlüsselwort aus der folgenden Liste als **x**-Wert festlegen: "**bottomright**", "**bottom**", "**bottomleft**", "**left**", "**topleft**", "**top**", "**topright**", "**right**", "**center**".
- **title**
  Eine Zeichenfolge mit dem Titel der Legende
- **legend**
  Zeichenfolgen-Vektor mit den Etiketten zu den Linien bzw. Symbolen

Sollen Linien erläutert werden, die sich nach Typ, Stärke und/oder Farbe unterscheiden, sind folgende Argumente zu verwenden:

- **lty**
  Ein Vektor mit den Linientypen
- **lwd**
  Ein Vektor mit den Linienstärken
- **col**
  Ein Vektor mit den Linienfarben

Sollen Datengruppen erläutert werden, die sich nach Symbol und/oder Farbe unterscheiden, sind folgende Argumente zu verwenden:

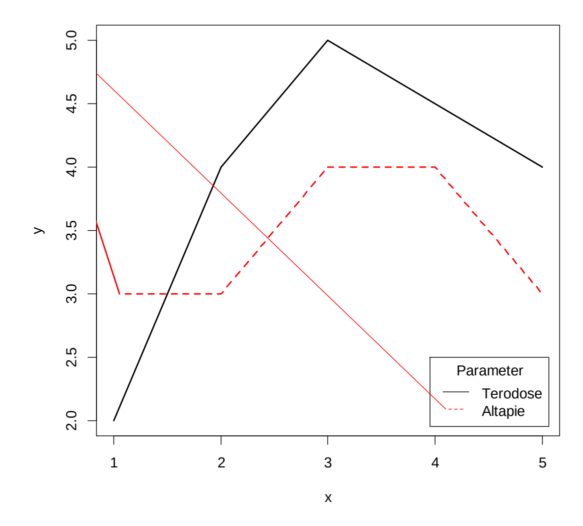
- **pch**
  Ein Vektor mit den Symbolen
- **col**
  Ein Vektor mit den Linienfarben

Um Datengruppen *und* Linien gemeinsam in einer Legende zu erläutern, können z.B. die Argumente **lty**, **pch** und **col** gemeinsam verwendet werden.

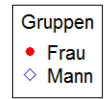
Das oben erstellte Liniendiagramm erhält durch den folgenden Funktionsaufruf

```
> legend(3.95,2.5, title="Parameter", legend=c("Terodose", "Altapie"),
+ lty=c(1,2), col=c("black", "red"))
```

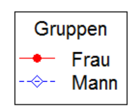
eine Legende:



In Abschnitt 9.2.4.2.3 wird für ein gruppiertes Streudiagramm eine Legende zur Erläuterung von Symbolen erstellt:



Nach Erweiterung dieses Streudiagramms um gruppenspezifische Regressionsgeraden werden in der Legende zu jeder Gruppe ein Symbol *und* eine Linie angezeigt:



Nach dem Kommando

```
> ?legend
```

liefert die **R**-Hilfe zahlreiche weitere Details zur Legendenbildung.

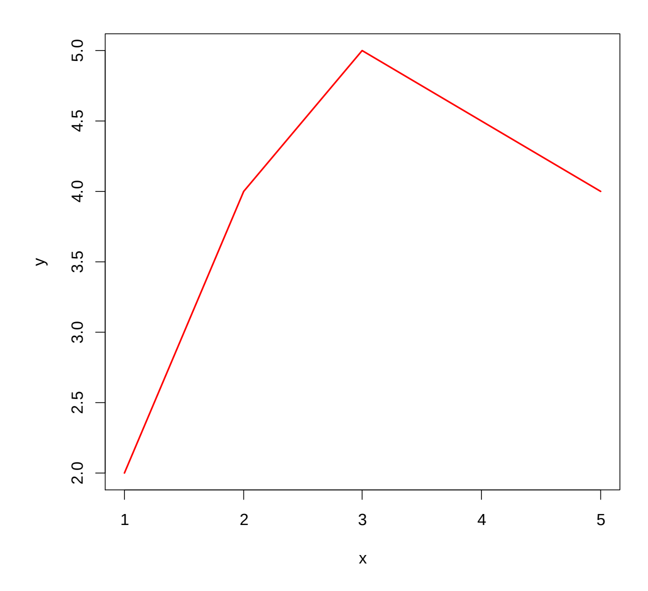
9.2.4 Wichtige Diagrammtypen

9.2.4.1 Liniendiagramm

Wenn sich bei einem **plot()** - Aufruf alle Punkte auf einer Linie befinden, und das **type**-Argument einen passenden Wert erhält (z.B.: `"l"`),

```
> x <- 1:5; y <- c(2, 4, 5, 4.5, 4)
> plot(x, y, type="l", col="red", lwd=2)
```

dann resultiert ein Liniendiagramm, z.B.:



9.2.4.2 Streudiagramm

In diesem Abschnitt nutzen wir die Funktion **plot()** dazu, Streudiagramme zu erstellen. Zunächst entstehen Varianten, die bei weitgehend äquivalentem Ergebnis auch mit SPSS möglich sind (bis Abschnitt 9.2.4.2.4). Schließlich werden die erlernten Techniken aber auch dazu verwendet, spezielle Streudiagramme zu produzieren, die mit SPSS (aktuell) weniger gut zu erstellen sind.

9.2.4.2.1 Einfaches Streudiagramm

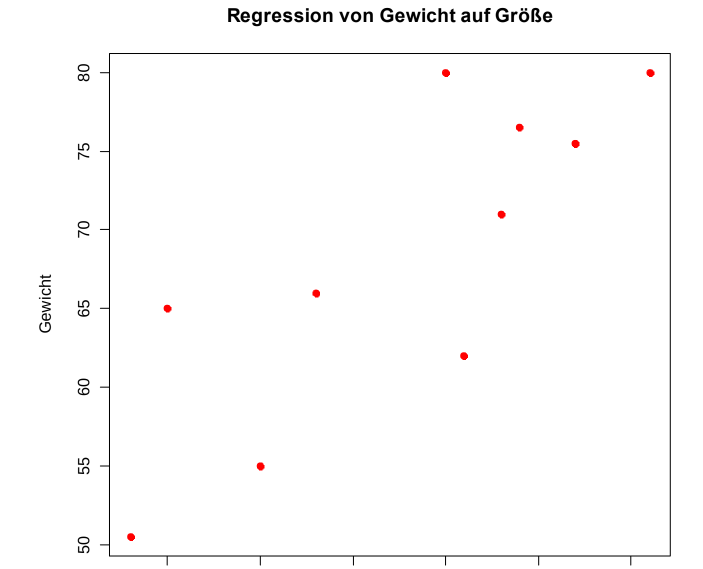
Die in Abschnitt 9.2.3 beschriebene **plot()** - Funktion erstellt beim voreingestellten Wert **p** für das Argument **type** ein zweidimensionales Streudiagramm für Punkte mit X- bzw. Y-Koordinaten in den Vektoren **x** und **y**, die als erstes bzw. zweites Argument zu übergeben sind, z.B.:

```
> groesse <- c(186,178,182,160,168,NA,165,179,158,175,176,176)
> gewicht <- c(80,71,75.5,65,66,76,55,76.5,50.5,80,62,NA)
> plot(groesse, gewicht, main="Regression von Gewicht auf Größe",
+ xlab="Größe", ylab="Gewicht", pch=19, col="red")
```

In diesem **plot()**-Aufruf werden zur Gestaltung des Diagramms einige Argumente verwendet, die in den Abschnitten 9.2.2 (über Grafikparameter und Beschriftungen) und 9.2.3 (über die **plot()** - Funktion) vorgestellt worden sind:

- **main**
  Hauptüberschrift
- **xlab, ylab**
  Achsenbeschriftungen
- **pch**
  Markierungsart für die Datenpunkte
- **col**
  Zeichenfarbe

Das Ergebnis:



R erlaubt zur Spezifikation der beiden Variablen auch die Modellsyntax (vgl. Abschnitt 8.2), wobei auf das Kriterium (die Y-Achsen-Variable) nach einer Tilde (~) der Regressor folgt, z.B.:

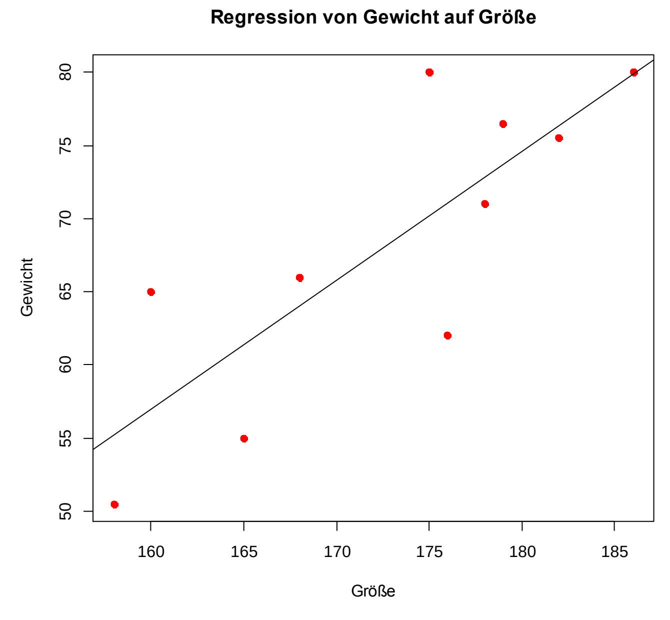
```
> plot(gewicht ~ groesse, main="Regression von Gewicht auf Größe",
+  xlab="Größe", ylab="Gewicht", pch=19, col="red")
```

9.2.4.2.2 Regressionsgerade und sonstige Linien ergänzen

Auf dem Weg zur Regressionsgeraden lassen wir die lineare Regression von Gewicht auf Größe durch die Funktion **lm()** schätzen (zur Modellformulierung siehe Abschnitt 8.2). Die Low-Level - Grafikfunktion **abline()**:

```
> abline(lm(gewicht ~ groesse))
```

zeichnet unter Verwendung der **lm()** - Ergebnisse die Regressionsgerade:



Zur Anzeige einer nichtlinearen (z.B. quadratischen) Anpassungslinie eignen sich die Funktionen **lines()** (vgl. Abschnitt 9.2.3) und **predict()**, z.B.:

```
> lines(groesse, predict(lm(gewicht~groesse+I(groesse^2))))
```

Als Y-Koordinaten werden die von **predict()** gelieferten Prognosen eines linearen Modells verwendet, das einen (im konkreten Beispiel eigentlich überflüssigen) quadratischen Term enthält. Im konkreten Fall klappt der **lines()** - Aufruf allerdings nicht, weil die **predict()** - Rückgabe und der Vektor `groesse` wegen eines fehlenden `groesse`-Werts eine verschiedene Länge haben. Um zu gültigen Argumenten für die Funktion **lines()** zu kommen, wird ein neuer Vektor mit Werten im Bereich des Regressors erzeugt:

```
> dfx <- data.frame(groesse = seq(155, 190, 1))
```

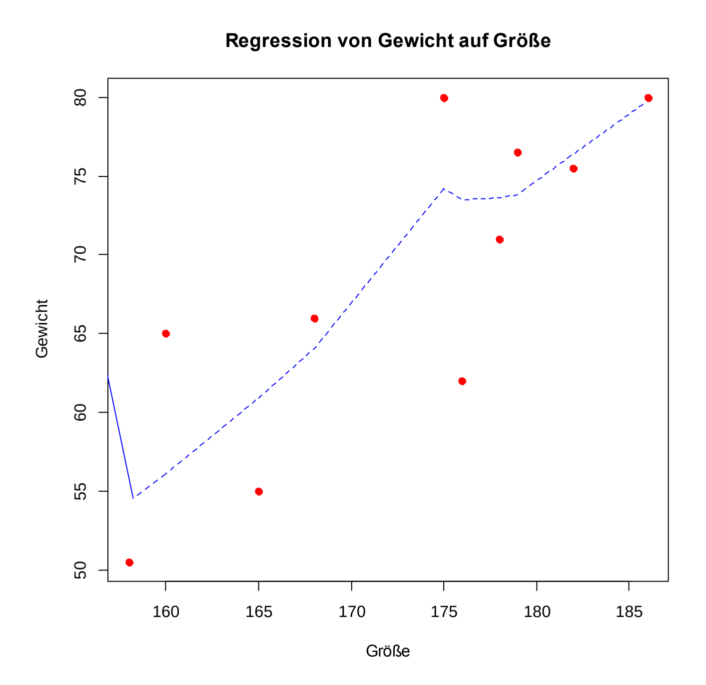
Auf den Vektor `dfx$groesse` (in der Datentabelle `dfx`) wird das Modell aus den realen Daten angewendet, und die **lines()** - Funktion erhält schließlich 2 gleichlange Vektoren:

```
> qmodGG <- lm(gewicht ~ groesse + I(groesse^2))
> lines(dfx$groesse, predict(qmodGG, newdata=dfx))
```

Mit Hilfe der Funktionen **lines()** und **lowess()** lässt sich eine lokal optimierte Modellprognose einzeichnen. Im Beispiel steigt der technische Aufwand etwas, weil die Vektoren `groesse` und `gewicht` jeweils einen **NA**-Wert enthalten. Alle unvollständigen Fälle müssen entfernt werden, was mit einer neuen Datentabelle und der **subset()** - Funktion (siehe Abschnitt 7.5) gelingt:

```
> daten <- data.frame(groesse, gewicht)
> daten <- subset(daten, !is.na(groesse) & !is.na(gewicht))
> lines(lowess(daten$gewicht ~ daten$groesse), lty=2, col="blue")
```

Das Ergebnis:



Mit der Low-Level - Grafikfunktion **abline()** kann man eine durch die Argumente **a** und **b** definierte Linie zeichnen, z.B. eine „normative“ Regressionsgerade zur Idealgewichtsempfehlung „Größe - 100 - 10%“, die in mathematischer Formulierung zu folgenden Koeffizienten führt:[1]

```
> abline(a=-90, b=0.9, col="blue")
```

Für das Ergänzen von horizontalen oder vertikalen Linien ist die Funktion **abline()** ebenfalls zuständig. Den Argumenten **h** und **v** ist als Wert die Achsenposition zu übergeben, z.B.:

```
> abline(h=68)
```

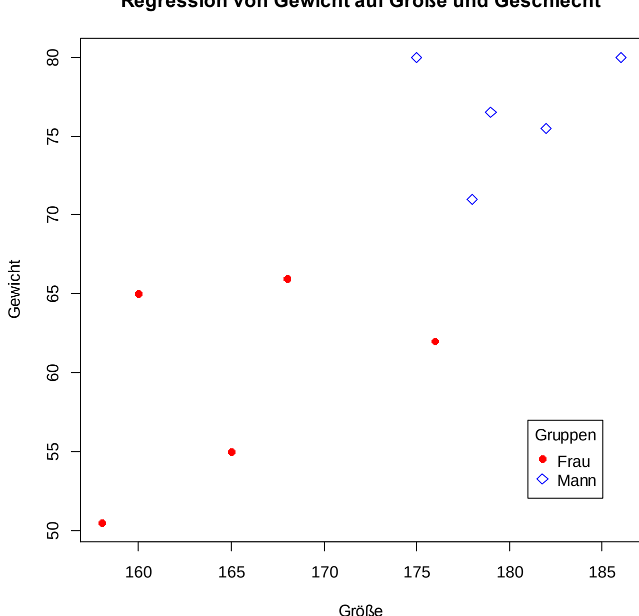
[1] „Größe - 100 - 10%“ = (Größe - 100) * 0,9 = -90 + 0,9 * Größe

9.2.4.2.3 Gruppiertes Streudiagramm

Mit der folgenden Anweisung ergänzen wir im aktuellen Bespiel mit den Variablen (numerischen Vektoren) `groesse` und `gewicht` noch einen Faktor mit dem Geschlecht der Probanden:

```
> geschl <- factor(c(2,2,2,1,1,1,1,2,1,2,1,2), labels=c("Frau", "Mann"))
```

Nun soll das folgende gruppierte Streudiagramm erstellt werden, das die Geschlechtszugehörigkeit anhand der Symbole erkennen lässt:



Über die in Abschnitt 7.3.1 beschriebene Rekodierung erstellen wir den Vektor `syms`, der für jeden Fall eine Symbolnummer enthält. Wir verwenden für Frauen das Symbol Nummer 16 (●) und für Männer das Symbol Nummer 5 (◇):

```
> syms <- numeric(length(geschl))
> syms[geschl=="Frau"] <- 16
> syms[geschl=="Mann"] <- 5
```

Analog gelangen wir zum **character**-Vektor `cols` mit der Farbe Rot für die Frauen und der Farbe Blau für die Männer:

```
> cols <- character(length(geschl))
> cols[geschl=="Frau"] <- "red"
> cols[geschl=="Mann"] <- "blue"
```

Mit den Vektoren `syms` bzw. `cols` als Werten für die Argumente `pch` bzw. `col` erstellen wir das Diagramm neu:

```
> plot(groesse, gewicht, main="Regression von Gewicht auf Größe und Geschlecht",
+ xlab="Größe", ylab="Gewicht", pch=syms, col=cols)
```

Schließlich erstellen wir noch eine Legende mit Hilfe der Low-Level - Grafikfunktion **legend()**:

```
> legend(181,57, title="Gruppen", legend=c("Frau", "Mann"),
+ pch=c(16,5), col=c("red", "blue"))
```

Wenn man damit einverstanden ist, die Symbole mit den Nummern 1 bzw. 2 für Frauen bzw. Männer zu verwenden (= interne Werte für die `geschl`-Faktorstufen) und auf Farben keinen Wert legt, dann lässt sich der Aufwand für die geschlechtsspezifischen Markierungen reduzieren:

```
> plot(groesse, gewicht, main="Regression von Gewicht auf Größe und Geschlecht",
+ xlab="Größe", ylab="Gewicht", pch=as.numeric(geschl))
```

Man erstellt aus dem `geschl`-Faktor mit der Funktion **as.numeric()** einen numerischen Vektor und verwendet diesen als Wert für das **plot()** - Argument **pch**.

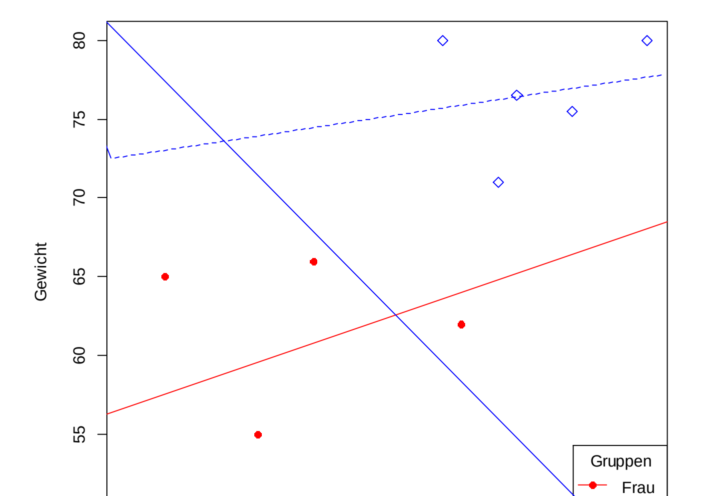
Um geschlechtsbedingte Regressionsgeraden einzuzeichnen, erweitert man den in Abschnitt 9.2.4.2.2 vorgestellten **abline()** - Funktionsaufruf um eine Fallauswahl über Indexvektoren:

```
> abline(lm(gewicht[geschl=="Frau"] ~ groesse[geschl=="Frau"]), lty = 1, col="red")
> abline(lm(gewicht[geschl=="Mann"] ~ groesse[geschl=="Mann"]), lty = 2, col="blue")
```

Mit einer um die Linientypen erweiterten Legende in der rechten unteren Ecke

```
> legend("bottomright", title="Gruppen", legend=c("Frau", "Mann"),
+        pch=c(16,5), lty=c(1,2), col=c("red", "blue"))
```

sieht das gruppierte Streudiagramm jetzt so aus:



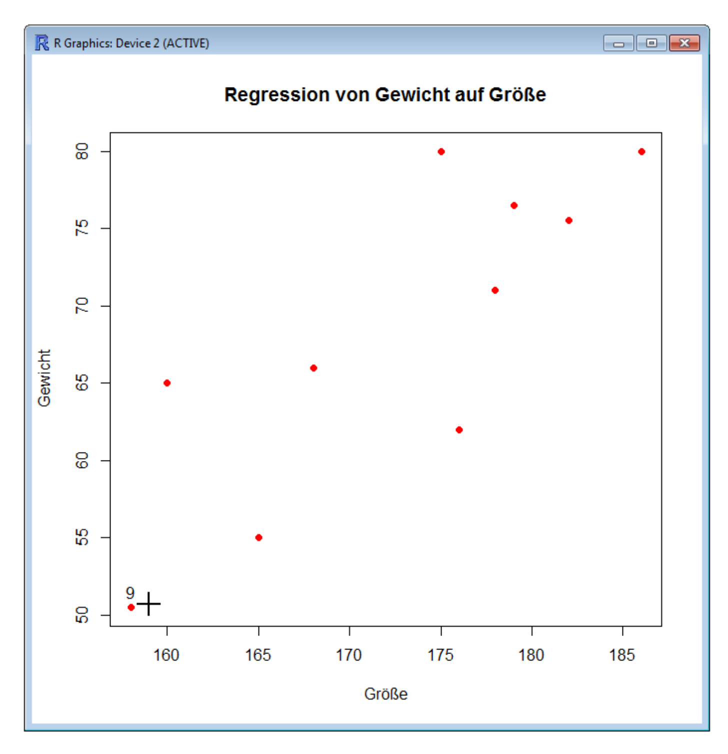
Das klassische Grafiksystems bietet noch weitere Optionen für Streudiagramme (z.B. Konfidenzintervalle, Streudiagramm-Matrizen), über die z.B. Muenchen (2011, S. 480ff) informiert.

9.2.4.2.4 Fallidentifikation

Befindet sich im aktiven Grafikfenster ein Streudiagramm mit den Variablen `groesse` und `gewicht`, dann kann man nach Ausführung der Anweisung **identify()** - Funktion

```
> identify(groesse, gewicht)
```

die Datenpunkte im Grafikfenster per Mausklick beschriften, z.B.:

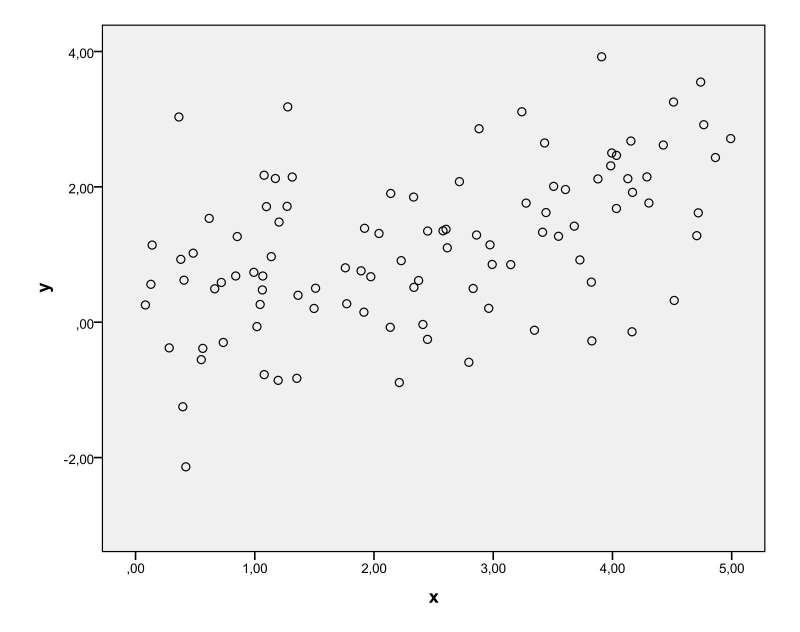


Um den Fallidentifikationsmodus zu beenden, ...

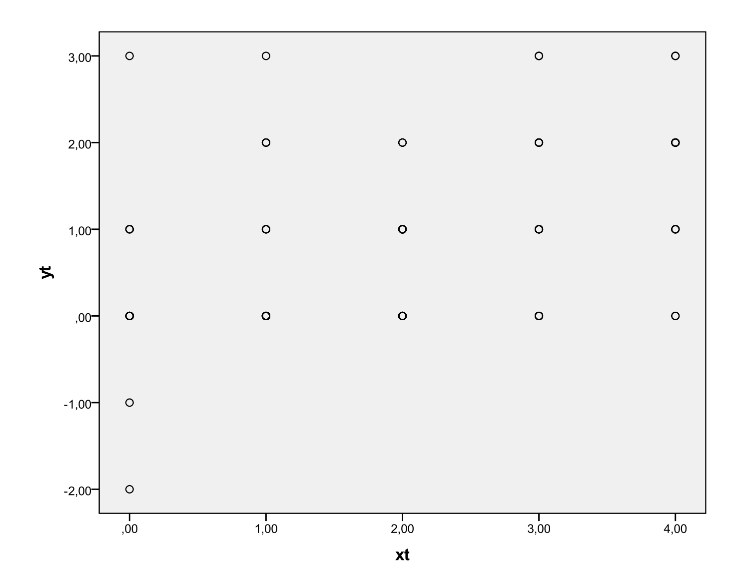
- drückt man die **Esc**-Taste
- oder wählt das Item **Stopp** aus dem Kontextmenü zum Grafikfenster.

9.2.4.2.5 Jitter-Darstellung

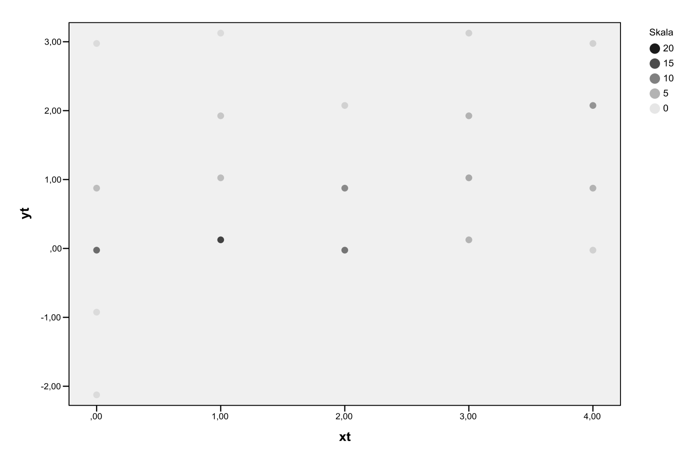
Das folgende (mit SPSS erstellte) Streudiagramm zeigt zwei deutlich miteinander korrelierte metrische Variablen:



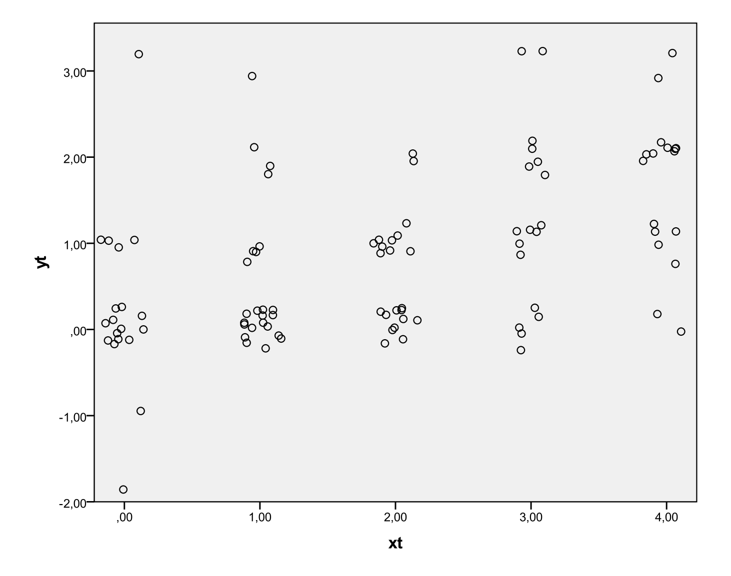
Sind statt der metrischen Variablen nur vergröberte Messungen vorhanden (mit wenigen Ausprägungen, vgl. Abschnitt 3.3), dann ergeben sich im Streudiagramm zahlreiche mehrfach besetzte Punkte, und die Interpretation ist erschwert, z.B.



Der Zusammenhang wird etwas deutlicher, wenn die Farbintensität der Punkte von der Häufigkeit abhängt:



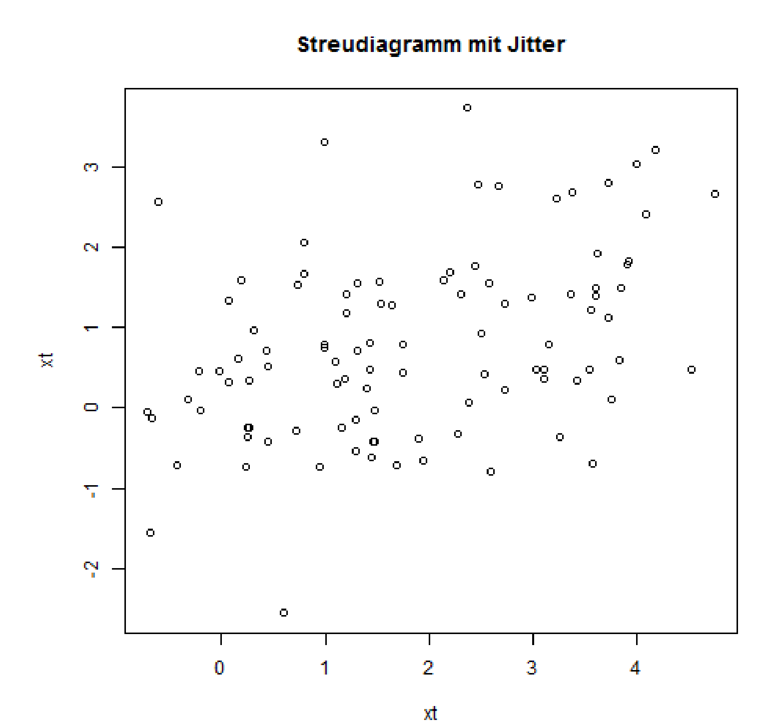
Bei der Jitter-Technik zur Lösung des Problems addiert man kleine Zufallsschwankungen zu den Beobachtungswerten, um die Punkte auseinander zu ziehen. In SPSS ist diese Technik mit Hilfe der GPL-Syntax realisierbar, doch lässt sich der Jitter-Grad nicht einstellen, so dass im Beispiel kein großer Nutzen entsteht:



In **R** erlaubt die **jitter()** - Funktion einen einstellbaren „Verwacklungsgrad“,

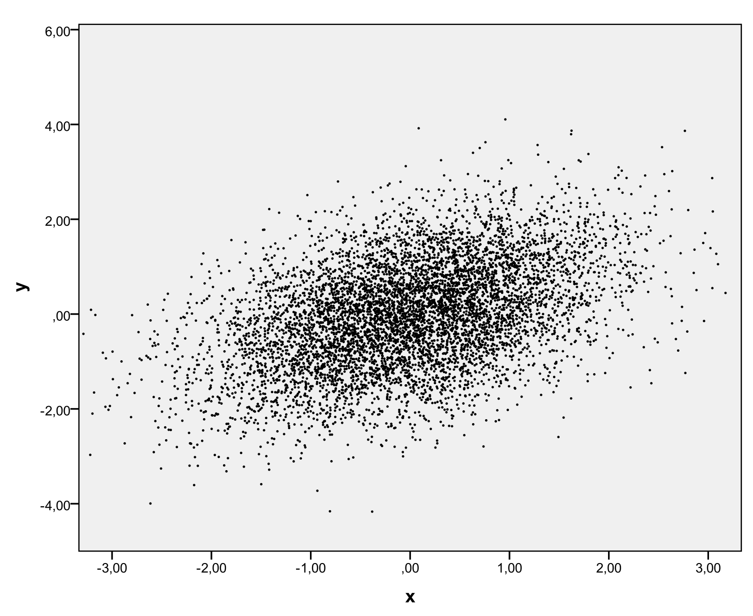
```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
plot(jitter(data$xt,4),jitter(data$yt,4),main="Streudiagramm mit Jitter",xlab="xt",ylab="xt")
END PROGRAM.
```

und die **plot()** - Funktion liefert mit den so behandelten Daten ein besser interpretierbares Bild:

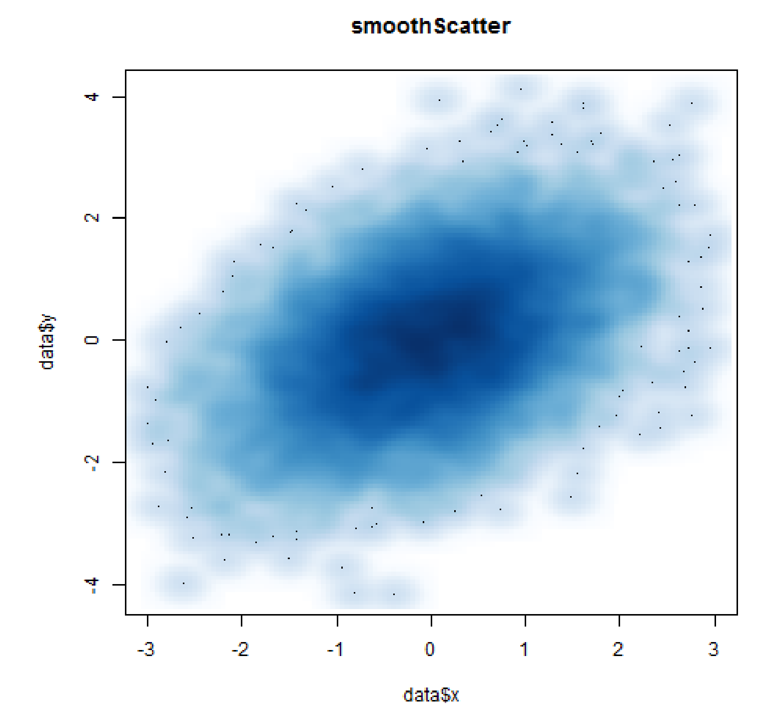


9.2.4.2.6 Farbliche Dichtedarstellung bei großen Stichproben

Sind sehr viele Datenpunkte vorhanden, ist ein Streudiagramm auch bei Wahl von winzigen Markierungen nicht ideal, z.B.:



Die **R**-Standardgrafik bietet für diesen Fall mit der Funktion **smoothScatter()** eine zumindest ästhetisch interessante Alternative, wobei die Wahrscheinlichkeitsdichte durch die Farbintensität ausgedrückt wird, z.B.:



Das Beispiel wurde über ein SPSS-Syntaxfenster angefordert:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
smoothScatter(data$x, data$y, xlim=c(-3,3), main="smoothScatter")
END PROGRAM.
```

9.2.4.3 Boxplot

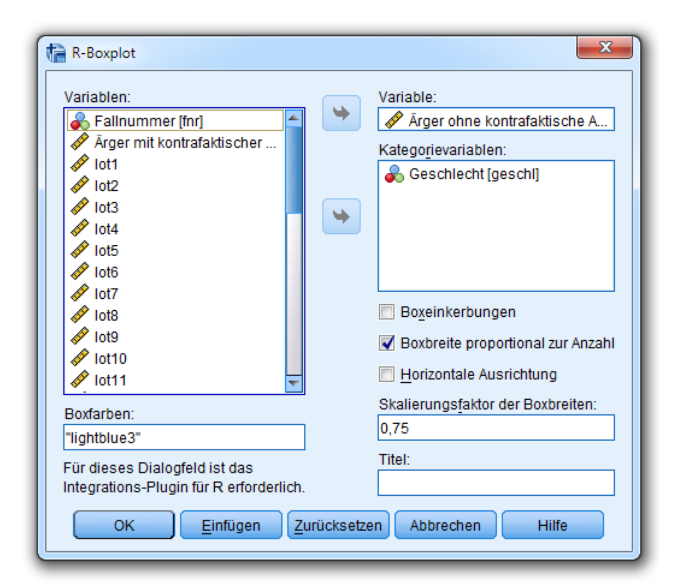
Zum Erstellen von Boxplots bieten **R** und SPSS weitgehend äquivalente Möglichkeiten:

- Darstellung einer einzelnen Variablen
- Mehrere Variablen nebeneinander
- Mehrere Gruppen nebeneinander (univariates ein- oder zweifaktorielles Design)

Sind in SPSS Statistics 22 die **R**-Essentials installiert (vgl. Abschnitt 2), dann stehen die wichtigsten Boxplot-Optionen der traditionellen **R**-Grafik in SPSS nach dem Menübefehl

Grafik > R-Boxplot

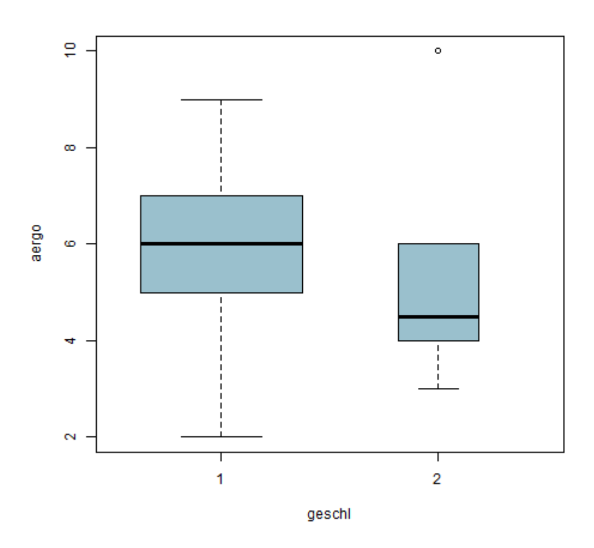
über die folgende Dialogbox zur Verfügung:



Hier bestehen u.a. die folgenden Einstellmöglichkeiten:

- **Boxbreite proportional zur Anzahl**
  Die Breite lässt sich zur Anzeige der Teilstichprobengröße nutzen.
- **Skalierungsfaktor der Boxbreiten**
  Damit lässt sich die generelle Breite der Boxen beeinflussen.
- **Boxfarben**
  Hier ist ein **R**-Farbname einzutragen.

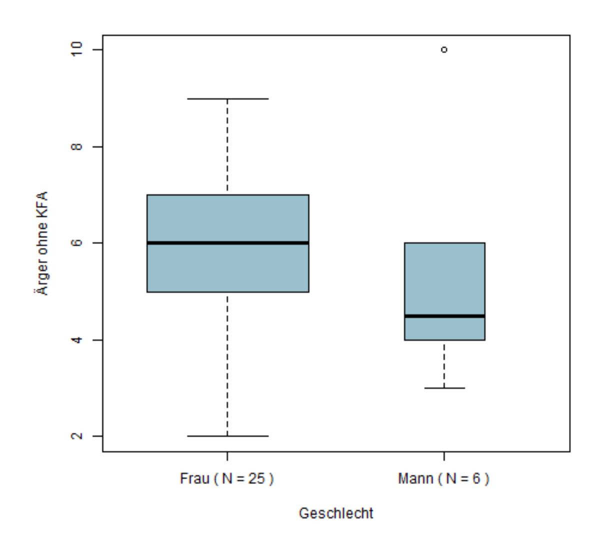
Das Ergebnis macht deutlich, dass die beiden Teilstichproben unterschiedlich groß sind, wobei aber exakte Angaben fehlen:



Bei direkter Nutzung der **R**-Funktion **boxplot()** über **R**-Anweisungen im SPSS-Syntaxfenster

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS(factorMode="labels")
levels(data$geschl)[1] <- paste(levels(data$geschl)[1], "(N = ",
                             length(data$geschl[data$geschl=="Frau"]), ")")
levels(data$geschl)[2] <- paste(levels(data$geschl)[2], "(N = ",
                             length(data$geschl[data$geschl=="Mann"]), ")")
boxplot(data$aergo ~ data$geschl, col="lightblue3", varwidth=TRUE, boxwex=0.75,
        xlab="Geschlecht", ylab="Ärger ohne KFA")
END PROGRAM.
```

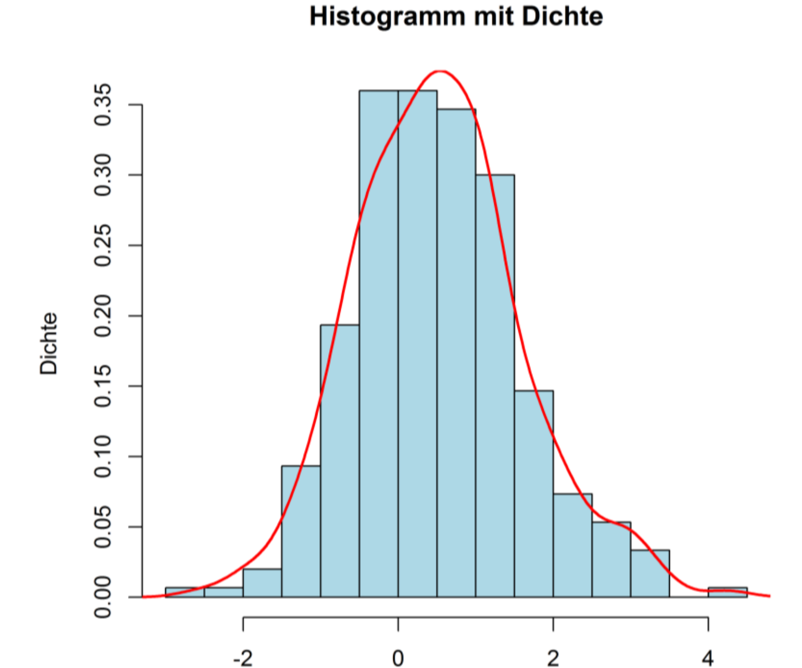
lässt sich mit Hilfe von Grafikparametern und **boxplot()** - Argumenten ein besseres Ergebnis erzielen:



Zur Anzeige der Fallzahlen wurden die Labels der Faktorstufen modifiziert, wobei die neuen Zeichenfolgen mit Hilfe der **R**-Funktion **paste()** aus den alten und den berechneten Teilstichprobenumfängen entstanden sind. Hier zeigt sich exemplarisch, wie mit **R**-Grafiken eine Verbesserung zu erzielen ist, die mit einem gewissen Aufwand bezahlt werden muss.

9.2.4.4 Histogramm mit Dichteschätzung

Mit den **R**-Funktionen **hist()**, **density()** und **lines()**
lässt sich (wie schon in Abschnitt 9.1.5 demonstriert) zu einer Variablen ein Histogramm mit geschätzter Dichtefunktion erstellen:



Diese Darstellung wurde für eine SPSS-Variable angefordert (vgl. Abschnitt 4.1) und mit publikationstauglicher Auflösung in eine PNG-Datei geschrieben:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
png("u:/eigene dateien/r/NormDens.png", 15, 15, units="cm", res=600)
hist(data$y, freq=FALSE, breaks=10, col="lightblue", main="Histogramm mit Dichte",
     ylab="Dichte", xlab=NA)
lines(density(data$y), col="red", lwd=2)
END PROGRAM.
```

Die **hist()** - Funktion gehört zu den High Level - Grafikfunktionen in **R**, die jeweils ein komplettes Diagramm produzieren. Im obigen **hist()** - Aufruf werden neben den generellen Grafikparametern **col**, **main**, **xlab** und **ylab** für die Zeichenfarbe bzw. für Beschriftungen (vgl. Abschnitt 9.2.2) folgende Argumente genutzt:

- **freq=FALSE**
  Statt absoluter Häufigkeiten sollen *relative* zur Beschriftung der Y-Achsen-Teilstriche verwendet werden.
- **breaks**
  Mit diesem Parameter beeinflusst man die Anzahl der Intervalle auf der X-Achse, wobei zu kleine oder zu große Werte zu einem wenig aussagekräftigen Histogramm führen.

Der Low Level - Grafikfunktion **lines()** - Funktion wird im Beispiel als erstes Argument die Funktion **density()** zur Definition des Linienverlaufs zu übergeben. Mit dem Grafikparameter **lwd** wird die Linienstärke beeinflusst.

9.2.4.5 Mehrere Diagramme kombinieren

Sollen zu Vergleichszwecken mehrere Diagramme in *einer* Abbildung kombiniert werden, definiert man mit dem Grafikparameter **mfrow** eine Platzierungsmatrix, deren Zellen durch die anschließenden Grafikfunktionsaufrufe gefüllt werden. Sollen z.B. zwei Diagramme übereinander erscheinen, verwendet man den folgenden Aufruf der **par()** - Funktion (vgl. Abschnitt 9.2.2):

```
> par(mfrow=c(2,1))
```

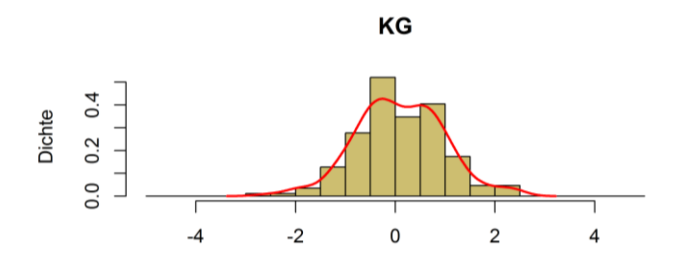
Zur Demonstration verwenden wir die künstlichen Daten aus Abschnitt 9.2.4.4 und fügen in SPSS noch die Variable `treat` zur Gruppeneinteilung hinzu, die einen Effekt auf die abhängige Variable `y` ausübt:

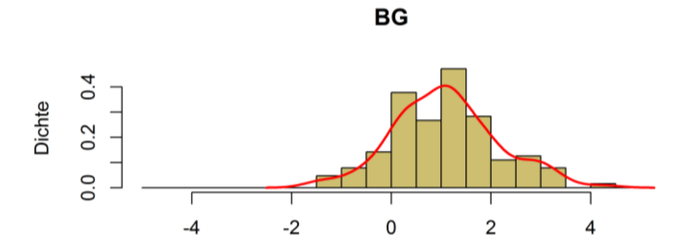
```
do if uniform(1) < 0.5.
  compute treat = 0.
else.
  compute treat = 1.
  compute y = y + 1.
end if.
```

Mit den folgenden, aus einem SPSS-Syntaxfenster abgeschickten **R**-Anweisungen werden zwei übereinander stehende Histogramme mit empirischer Dichte erstellt:

```
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
g0 <- data$y[data$treat==0]
g1 <- data$y[data$treat==1]
png("u:/eigene dateien/r/NormDens2.png", 15, 15, units="cm", res=600)
par(mfrow=c(2,1))
hist(g0, freq=FALSE, breaks=seq(-5,5,0.5), col="lightgoldenrod3", main="KG", xlab="", ylab="Dichte")
lines(density(g0), col="red", lwd=2)
hist(g1, freq=FALSE, breaks=seq(-5,5,0.5), col="lightgoldenrod3", main="BG", xlab="", ylab="Dichte")
lines(density(g1), col="red", lwd=2)
par(mfrow=c(1,1))
END PROGRAM.
```

Wegen der identischen Klassendefinitionen (Argument **breaks**) wird der „Behandlungseffekt“ gut sichtbar:



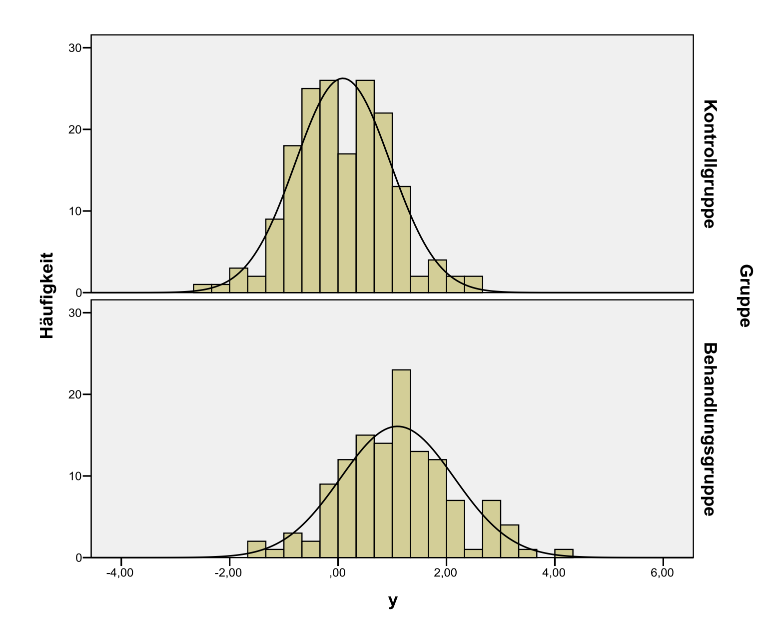


Am Ende wird der Grafikparameter **mfrow** wieder auf seinen Standardwert **c(1,1)** zurückgesetzt.

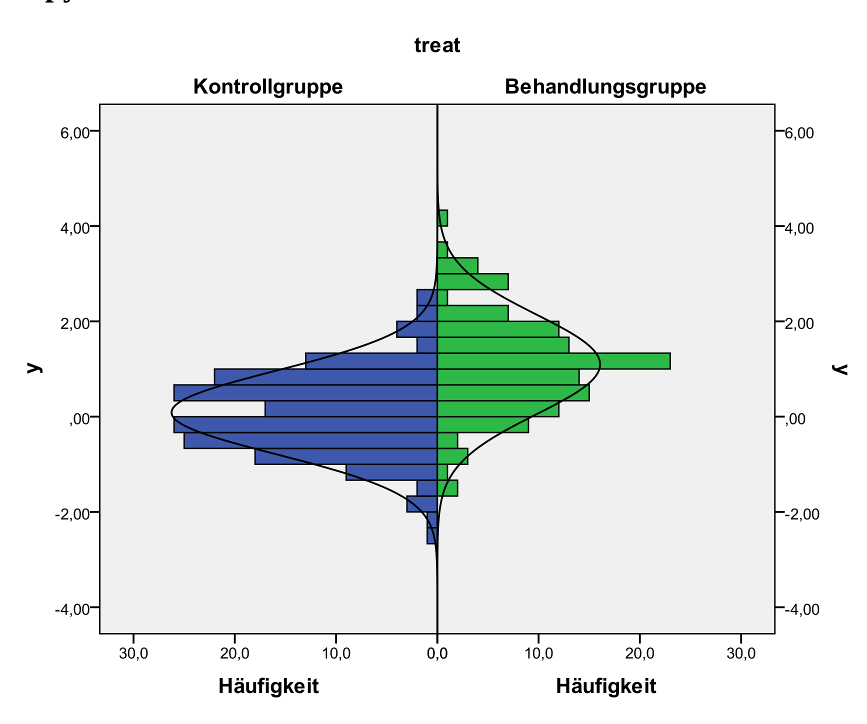
In SPSS lässt sich mit dem Kommando

```
GRAPH
  /HISTOGRAM(NORMAL)=y
  /PANEL ROWVAR=treat ROWOP=CROSS.
```

ein analoges Diagramm erstellen, wobei die Dichteschätzung eine Normalverteilung voraussetzt:



Als alternative Darstellung zum Vergleich von zwei Histogrammen bietet die Diagrammerstellung von SPSS die **Populationspyramide** an:

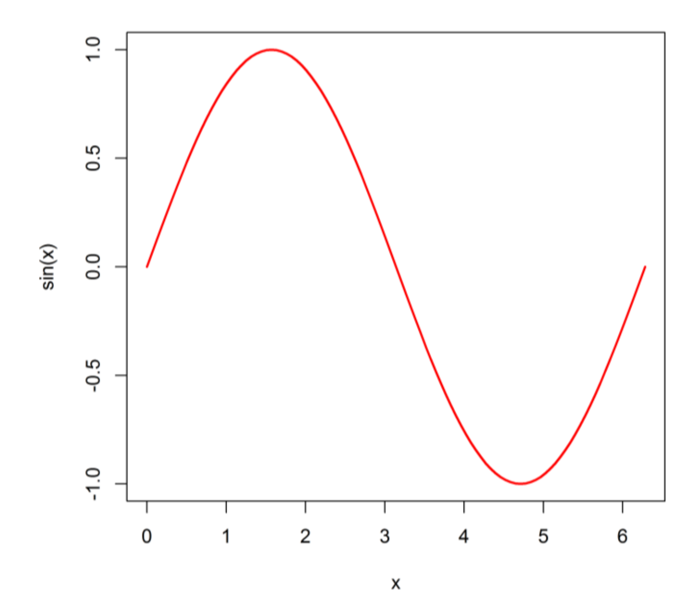


9.2.4.6 Funktionsplots

Mit der High Level - Grafikfunktion **curve()** kann man den Graphen einer Funktion von einer Veränderlichen plotten, z.B. den Sinus im Intervall [0, 2π]:

```
> curve(sin(x), 0, 2*pi, col="red", lwd=2)
```

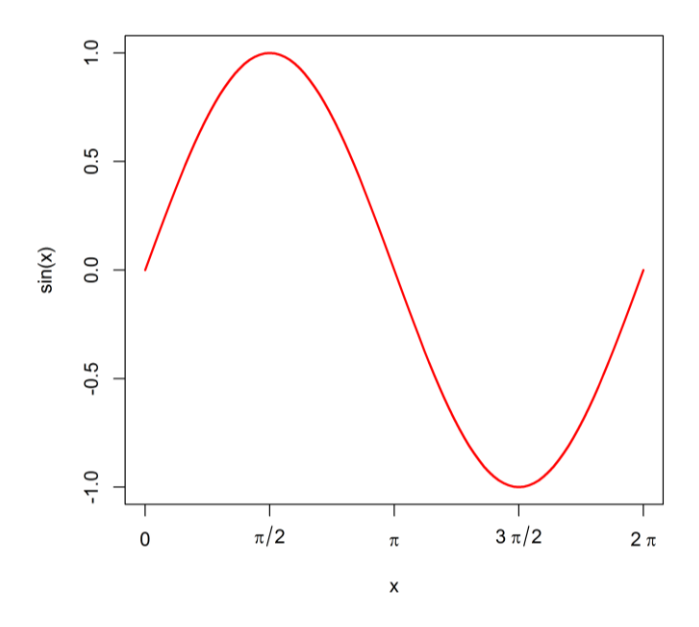
Beim Ergebnis des ersten Versuchs stört, dass die X-Achsenbeschriftung keinen Bezug zu den kritischen Punkten im Sinus-Verlauf hat:



Daher schalten wir im **curve()** - Aufruf die X-Achse aus und erzeugen über die Low-Level - Grafikfunktion **axis()** (siehe Abschnitt 9.2.3.2) eine Achse mit Markierungen an den passenden Stellen (0, π/2, π, 3π/2, 2π):

```
> curve(sin(x), 0, 2*pi, col="red", lwd=2, xaxt="n")
> axis(side=1, at=c(0, pi/2, pi, 3*pi/2, 2*pi),
+ labels=c("0", expression(pi/2), expression(pi), expression(3~pi/2),
+ expression(2~pi)))
```

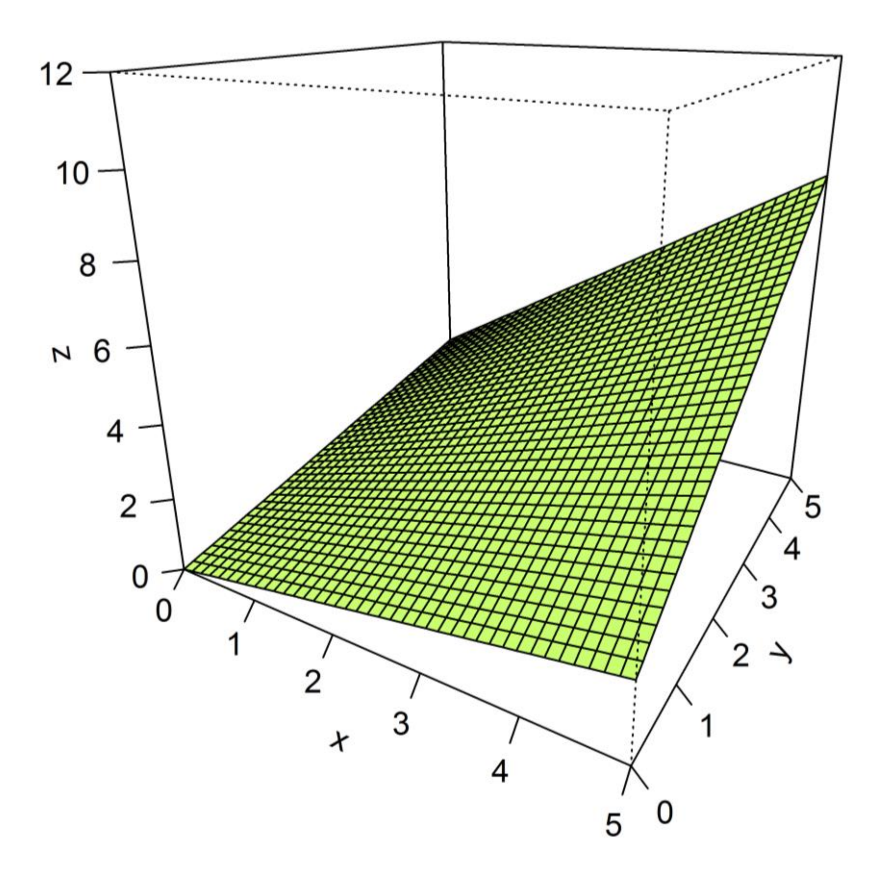
Das **labels**-Argument erhält einen **character**-Vektor mit den gewünschten Teilstrichbeschriftungen, wobei durch **expression()**-Aufrufe für die mathematische Typographie gesorgt wird:



Als Demonstration für einen 3D-Plot soll die Reaktionsoberfläche einer multiplen Regression mit zwei interagierenden metrischen Regressoren (siehe Baltes-Götz 2014b) mit der Funktion **persp()** gezeichnet werden:

```
> png("u:/eigene dateien/r/ModReg.png", 15, 15, units="cm", res=600)
> x <- seq(0, 5, length=40)
> y <- x
> f <- function(x, y) {
>   0.4*x + 0.4*y + 0.2*x*y
> }
> z <-outer(x, y, f)
> persp(x, y, z, zlim=range(0, 12), theta=30, phi=20, col="darkolivegreen1",
+       ticktype="detailed")
> dev.off()
```

Weil für das PNG-Ausgabegerät eine relativ hohe Auflösung gewählt wurde, ist das Ergebnis recht ansehnlich:



9.3 Das Grafikpaket ggplot2

Das von Wickham (2009) erstellte **R**-Paket **ggplot2** basiert auf der von Wilkinson (2005) entwickelten *Grammar of Graphics*. Seine anspruchsvolle, aber gut durchdachte Konstruktion macht die Anweisungen zur Definition von Diagrammen relativ übersichtlich. Im Ergebnis ist im Vergleich zur traditionellen Grafik bei einfachen Diagrammen der Aufwand etwas höher, bei komplexen Diagrammen aber deutlich geringen.

Weil die GG-Grafik nicht mit der traditionellen Technik arbeitet, sind die in Abschnitt 9.2.2 beschriebenen Grafikparameter hier bedeutungslos.

Um die Funktionen in **ggplot2** nutzen zu können, müssen Sie das Paket einmalig installieren:

```
> install.packages("ggplot2")
```

Neben dem Paket **ggplot2** werden in der Regel ca. 10 weitere, von **ggplot2** vorausgesetzte Pakete installiert. Vor jeder Verwendung muss **ggplot2** geladen werden:

```
> library(ggplot2)
```

Das Paket **ggplot2** bietet zum Erstellen eines Diagramms zwei Optionen:

- Den einfachen Weg über die Funktion **qplot()**
  Bei den Argumenten und Voreinstellungen dieser auf einfache Bedienbarkeit getrimmten Funktion (*Quick-Plot*) hat man sich an der traditionellen Graphik-Funktion **plot()** orientiert.
- Den flexiblen Weg über die Funktion **ggplot()**
  Dabei wird ein Diagramm inkrementell aus Schichten aufgebaut, was für große Flexibilität bei der Erstellung individueller Lösungen sorgt.

Es ist durchaus möglich, ein Diagramm-Objekt mit **qplot()** zu initialisieren und mit der Flexibilität mit **ggplot()** weitere Schichten zu ergänzen. Im Manuskript wird bevorzugt die Funktion **ggplot()** verwendet.

Derzeit beschränkt sich das Paket **ggplot2** auf 2D-Darstellungen, so dass sich z.B. keine Reaktionsoberflächen von Funktionen darstellen lassen (vgl. Abschnitt 9.2.4.6). Auf 3D-Effekte in genuin zweidimensionalen Diagrammen (z.B. in einem Balkendiagramm zur Verteilung einer nominalskalierten Variablen) wird bewusst verzichtet.

Zur vertieften und systematischen Einarbeitung in die Verwendung des **ggplot2** - Pakets eignen sich die folgenden Bücher:

- In Wickham (2009) liegt der Schwerpunkt auf der Logik der Grafikproduktion, wobei aber auch die Anwendungsmöglichkeiten demonstriert werden.
- Chang (2013) liefert mit seinem „Kochbuch“ Lösungen für typische Aufgaben.

Eine vollständige technische Dokumentation ist hier zu finden:

http://docs.ggplot2.org/current/

Es ist auf eine aktuelle Dokumentation zu achten, weil aufgrund der dynamischen Weiterentwicklung des **ggplot2**-Pakets so manches Beispiel aus einem älteren Lehrbuch nicht mehr klappt.

9.3.1 Grammatik eines ggplot2-Plots

9.3.1.1 Plot-Objekte, Schichten und Geome

Ein **ggplot2**-Plot besteht aus übereinander liegenden **Schichten**. Eine Schicht enthält eine Datenvisualisierung, die als **Geom** (*geometric object*) bezeichnet wird. Ein Geom kann z.B. (x,y) - Datenpunkte, Balken oder Linien enthalten.

Bevor der Schichtenausbau losgehen kann, muss ein **Plot-Objekt** angelegt werden. Dazu rufen wir die Funktion **ggplot()** auf und benennen im ersten Argument eine voreingestellte Datentabelle mit den darzustellenden Variablen. Sind Variablen auf mehreren Diagrammschichten mit bestimmten Darstellungsaspekten (z.B. mit der X- oder Y-Position) verbunden, gibt man diese Verbindung schon bei der Plot-Initialisierung bekannt, wobei ein Aufruf der Funktion **aes()** zu verwenden ist (vgl. Abschnitt 9.3.1.2). Im folgenden Beispiel werden die Variablen `größe` bzw. `gewicht` als erstes bzw. zweites unbenanntes Argument der **aes()**-Funktion der X- bzw. Y-Position des Diagramms zugeordnet:

```
> sd <- ggplot(ggg, aes(größe, gewicht))
```

Wird bei der Plot-Initialisierung darauf verzichtet, eine voreingestellte Datentabelle bzw. Rollenzuweisungen für Variablen zu benennen, müssen diesen Angaben später im Zusammenhang mit Schichtdefinitionen nachgeholt werden.

Wie ein (impliziter) **print()** - Aufruf für das eben erzeugte Plot-Objekt zeigt,

```
> sd
```

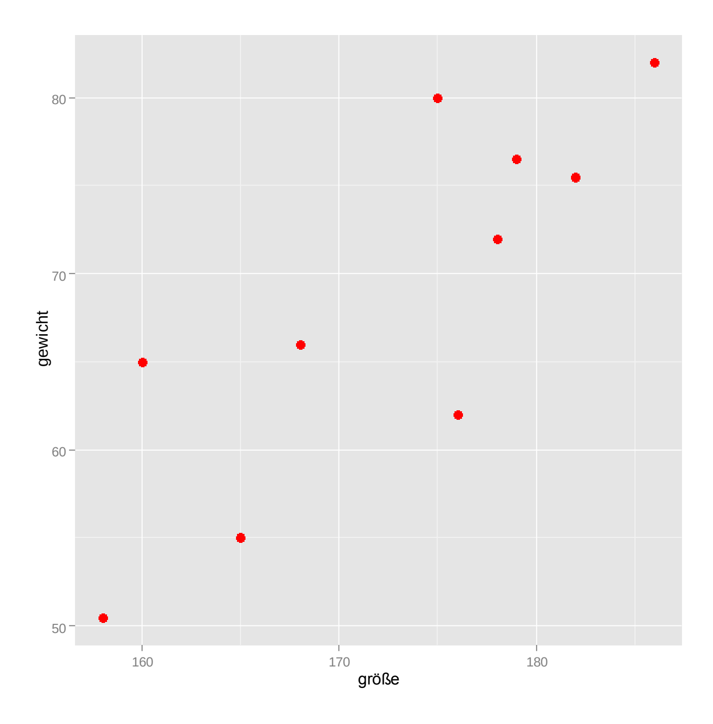
ist mangels vorhandener Schichten noch keine Grafikproduktion möglich:

```
Fehler: No layers in plot
```

Um für ein Plot-Objekt eine Schicht mit einem bestimmten Geom zu erstellen, ruft man eine Funktion mit der gewünschten Darstellungsart im Namen auf, z.B. **geom_point()** für eine Schicht mit Datenpunkten, die (x,y) - Wertepaare darstellen. Die Rückgabe der Schicht-Funktion wird mit dem Operator **+** zum Plot-Objekt hinzugefügt:

```
> sd + geom_point(colour="red", size=3)
```

Optional kann man mit ihren Argumenten z.B. schichtspezifische Abbildungen der ästhetischen Attribute des Geoms (siehe unten) auf Variablen oder feste Werte sowie eine schichtspezifische Datentabelle festlegen, wenn keine passenden Voreinstellungen auf Plot-Ebene bestehen. Im Beispiel erhalten wir folgendes Ergebnis:



Ein **ggplot2**-Objekt hat den Datentyp *Liste*, z.B.:

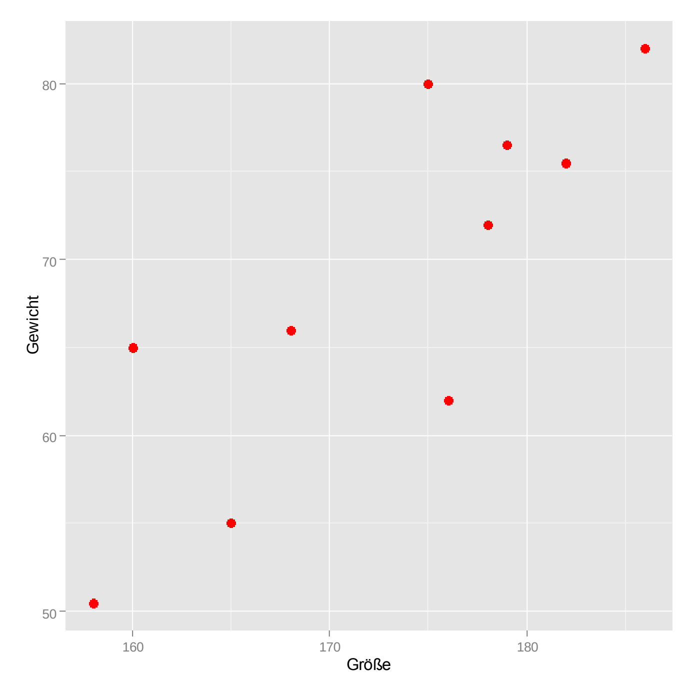
```
> str(sd)
List of 9
 $ data       :'data.frame':    12 obs. of  3 variables:
  ..$ geschlecht: Factor w/ 2 levels "Frau","Mann": 2 2 2 1 1 1 1 2 1 2 ...
  ..$ größe     : num [1:12] 186 178 182 160 168 NA 165 179 158 175 ...
  ..$ gewicht   : num [1:12] 82 72 75.5 65 66 76 55 76.5 50.5 80 ...
 $ layers     : list()
 $ scales     :Reference class 'Scales' [package "ggplot2"] with 1 fields
  ..$ scales: NULL
  ..and 21 methods, of which 9 are possibly relevant:
  ..  add, clone, find, get_scales, has_scale, initialize, input, n, non_position_scales
 $ mapping    :List of 2
  ..$ x: symbol größe
  ..$ y: symbol gewicht
 $ theme      : list()
 $ coordinates:List of 1
  ..$ limits:List of 2
  .. ..$ x: NULL
  .. ..$ y: NULL
  ..- attr(*, "class")= chr [1:2] "cartesian" "coord"
```

```
$ facet      :List of 1
 ..$ shrink: logi TRUE
 ..- attr(*, "class")= chr [1:2] "null" "facet"
$ plot_env   :<environment: R_GlobalEnv>
$ labels     :List of 2
 ..$ x: chr "größe"
 ..$ y: chr "gewicht"
- attr(*, "class")= chr [1:2] "gg" "ggplot"
```

Es ist durchaus möglich, einzelne Bestandteile dieser Liste (z.B. die Achsenbeschriftungen) zu modifizieren:

```
> sd$labels$x<-"Größe";sd$labels$y<-"Gewicht";sd+geom_point(colour="red",size=3)
```

und das Ergebnis auszugeben:



Wir lernen allerdings in Abschnitt 9.3.1.4.2 eine einfachere Methode zur Änderung der Achsenbeschriftungen kennen.

Weitere Beispiele für Geome bzw. generierende Funktionen:

- **geom_histogram()** erstellt ein Histogramm
- **geom_line()** erstellt einen Linienverlauf
- **geom_boxplot()** erstellt Boxplots für die Kategorien eines Faktors
- **geom_text()** erstellt Beschriftungen (z.B. zu den Datenpunkten eines Streudiagramms)

Auf der Webseite http://docs.ggplot2.org/current/ werden zur Version 0.9.3.1 von **ggplot2** 37 Geome beschrieben.

Ein Diagramm muss mindestens eine Schicht (ein Geom) enthalten und kann beliebig komplex aufgebaut sein. Man verwendet den Operator **+**, um Schichten zu ergänzen.

9.3.1.2 Aesthetics

Ein Geom besitzt visuelle Attribute (sogenannte **Aesthetics**), die jeweils auf eine Variable oder auf einen konstanten Wert abgebildet werden. Das Geom mit den (x,y) - Datenpunkten eines Streudiagramms unterstützt z.B. die folgenden Aesthetics (mit den **ggplot2**-Namen in Klammern), wobei die beiden ersten Attribute obligatorisch zu versorgen sind:

- X-Koordinaten der Datenpunkte (**x**)
- Y-Koordinaten der Datenpunkte (**y**)
- Form der Datenpunkte (**shape**)
- Farbe der Datenpunkte (**colour**)
- Transparenzgrad der Datenpunkte (**alpha**)
- Größe der Datenpunkte (**size**)

Auch jedes andere Geom besitzt eine Menge von *unterstützten* sowie eine Menge von *obligatorischen* Attributen.

Eine Liste mit allen vom Paket **ggplot2** unterstützten Geomen mitsamt den jeweiligen verfügbaren bzw. obligatorischen ästhetischen Attributen findet sich z.B. in Wickham (2009, Tabelle 4.3 in Abschnitt 4.6). Selbstverständlich kann man sich auch über die **R**-Hilfe informieren und z.B. mit

```
> ?geom_point
```

u.a. die Attribute zum Streudiagramm abrufen.

Die Datentabelle mit den zu visualisierenden Variablen sowie die Attribut-Abbildungen können auf Plot-Ebene festgelegt werden und sind dann für alle enthaltenen Schichten bzw. Geome gültig. Im folgenden Beispiel wird die **ggplot()**-Funktion (vgl. Abschnitt 9.3.2) benutzt, um ein Plot-Objekt anzulegen:

```
> sd <- ggplot(ggg, aes(größe, gewicht))
```

So vermeidet man es, die Spezifikation für mehrere Schichten wiederholen zu müssen. Wird ein Attribut auf Plot- und Schichtebene abgebildet, dann dominiert die lokale Zuordnung.

Ein ästhetisches Attribut kann auf einen festen Wert abgebildet werden, z.B. die Form von (x,y) - Datenpunkten:

```
> sd + geom_point(shape=3)
```

Weit wichtiger für den intendierten Zweck eines Diagramms ist es jedoch, ästhetische Attribute auf zu visualisierende Variablen abzubilden. Dazu ist die Funktion **aes()** zu verwenden, die als Argumente eine Liste von (Aesthetic = Variable) - Zuordnungen erhält, wobei auch Funktionen von Variablen erlaubt sind. Im folgenden Beispiel erzeugt die Funktion **geom_smooth()** zu einem gruppierten Streudiagramm eine Schicht mit Anpassungsfunktion und Konfidenzstreifen, wobei die Füllfarbe von der Variablen `geschlecht` abhängig gemacht wird, während der Transparenzgrad datenunabhängig festgelegt wird:

```
> sd + geom_point() + geom_smooth(method="lm",aes(fill=geschlecht),alpha=0.1)
```

Während die konstante Wertzuweisung nur für Schichten bzw. Geome möglich ist, kann die Variablen-gebundene, via **aes()** realisierte Zuweisung auf eine Schicht oder das gesamte Plot-Objekt angewendet werden.

Häufig arbeiten die Geome zu den Schichten eines Diagramms mit der voreingestellten (auf Plot-Ebene) vereinbarten Datentabelle und den dazu definierten Abbildungen von Attributen (Aesthetics) auf Variablen. Es ist jedoch erlaubt, dass ein Geom eine eigene Datentabelle mit eigenen Aesthetics-Abbildungen verwendet.

9.3.1.3 Statistische Transformationen

Für eine Schicht bzw. das dort präsentierte Geom kann eine **statistische Transformation** (Kurzbezeichnung: **stat**) der darzustellenden Variablen erforderlich sein. Im eben erwähnten Beispiel mit **geom_smoth()** als generierender Funktion entsteht eine neue Datentabelle, indem für gleichabständige Stützstellen die Werte einer Anpassungsfunktion ermittelt werden. Ein anderes Beispiel ist die Bildung von Intervallen (**stat = "bin"**) für ein Histogramm. Wird die identische Transformation als **stat**-Spezialfall einbezogen, kann man sagen, dass zu jedem Geom eine statistische Transformation gehört

(siehe Tabelle 4.3 in Abschnitt 4.6 von Wickham 2009). Außerdem hat jedes Geom eine voreingestellte statistische Transformation, und zu jeder Transformation gehört ein voreingestelltes Geom.

Mit der Funktion **args()** lässt sich die Voreinstellung für ein Geom bzw. eine statistische Transformation in Erfahrung bringen, z.B.:

```
> args(geom_bar)
function (..., stat = "bin", ...)

> args(stat_bin)
function (... , geom = "bar", ...)
```

Durch eine Änderung der Voreinstellung sind eigenständige Diagrammkreationen möglich.

Bei der Erstellung einer Schicht kann man sich zwischen zwei letztlich äquivalenten Vorgehensweisen entscheiden:

- Man verwendet eine **geom**-Funktion und spezifiziert nötigenfalls die statistische Transformation per **stat**-Argument, z.B. (vgl. Abschnitt 9.3.4.3.2)
  ```
  > ggplot(ggg,aes(x=geschlecht,y=gewicht))+geom_bar(stat="summary",fun.y=mean)
  ```
- Man verwendet eine **stat**-Funktion und spezifiziert nötigenfalls das gewünschte Geom per **geom**-Argument. Das Resultat aus dem letzten Beispiel lässt sich auch so anfordern:
  ```
  > ggplot(ggg,aes(x=geschlecht,y=gewicht))+stat_summary(geom="bar",fun.y=mean)
  ```

Ist eine statistische Transformation zu konfigurieren, werden die fälligen Argumente für die zuständige Funktion (z.B. **stat_bin()** bei einer Klassenbildung) an die generierende Geom-Funktion (z.B. **geom_histogram()**) übergeben und durchgeschleust, z.B.:

```
> ggplot(ggg, aes(gewicht)) + geom_histogram(binwidth = 10)
```

Sollte unklar sein, welche Argumente die zu einem Geom gehörige statistische Transformation unterstützt, kann man ...

- per **args()**-Funktion die voreingestellte statistische Transformation des Geoms ermitteln (siehe oben)
- und per Hilfesystem die Argumente der Transformationsfunktion anzeigen lassen (z.B. **?stat_bin**).

Gelegentlich muss man von einer statistischen Transformation gelieferte Variablen explizit ansprechen, z.B. einem ästhetischen Attribut zuweisen. Das geschieht im folgenden Beispiel mit der Variablen **density**, welche von der bei **geom_histogram()** voreingestellten Transformation **stat_bin()** produziert wird (vgl. Abschnitt 9.3.4.1). Um Verwechslungen mit Dataframe-Variablen zu vermeiden, sind vor und hinter die Namen von Transformationsergebnissen jeweils zwei Punkte zu setzen. Im folgenden Beispiel wird die Y-Achsenwert auf das Transformationsergebnis **density** abgebildet, so dass auf der Y-Achse relative statt absoluter Häufigkeiten angezeigt werden:

```
> ggplot(ggg,aes(gewicht)) + geom_histogram(binwidth=10,aes(y=..density..))
```

9.3.1.4 Skalen, Achsen und Legenden

Bei der Abbildung von Variablen auf ästhetische Attribute sind Skalen und das verwendete Koordinatensystem relevant:

- **Skalen**
  Rohwerte (z.B. Größenangaben in cm) müssen auf Werte abgebildet werden, die das Grafiksystem verarbeiten kann. Weil die **ggplot2**-Grafik auf dem **grid**-Paket basiert, sind z.B. für die X- bzw. Y-Positionen Werte im Intervall [0, 1] erlaubt. Eine Skalierung sorgt für die Abbildung der Rohwerte auf Werte, die das Grafiksystem versteht. Skalen sind nicht nur bei X- und Y-Koordinaten beteiligt, sondern auch bei anderen ästhetischen Attributen (Farbe, Form, Größe, Linientyp).
- **Koordinatensystem**
  Meist verwendet man das kartesische Koordinatensystem mit Achsen im Winkel von 90°. Gelegentlich kommen Alternativen wie Polarkoordinaten in Betracht.

Von den Skalen hängen auch die Achsen und die Legende eines Diagramms ab, die gemeinsam als **Guides** bezeichnet werden. Eine Skala muss *alle* Schichten eines Diagramms berücksichtigen.

Zur Modifikation von Skalen dienen Funktionen mit einem Namen nach dem folgenden Schema:

scale_<aesthetic>_<type>

Als **type**-Werte sind z.B. **discrete**, **continuous** und **gradient** erlaubt.

Den **scale**-Funktionen sind die folgenden Argumente gemeinsam:

- **name**
  Mit dem ersten Argument kann man den Titel einer Achse (**x** oder **y** als Aesthetic) oder einer Legende (**colour**, **fill**, **size**, **shape** oder **linetype** als Aesthetic) festlegen.
- **limits**
  Minimaler und maximaler zu berücksichtigender Wert. Fälle außerhalb dieser Grenzen werden ausgeschlossen, was sich auch auf andere Schichten (z.B. mit einer Regressionsfunktion) auswirkt.
- **breaks**
  In einem Vektor liefert man die Hauptunterteilungspunkte. Diese erhalten Etiketten, während die bei einer kontinuierlichen Skala per Voreinstellung mittig zwischen zwei Hauptunterteilungen eingefügten Nebenunterteilungen ohne Etikett bleiben.
- **labels**
  Ein Vektor mit den Etiketten zu den Hauptunterteilungspunkten.

Im folgenden Streudiagramm

```
> sd <- ggplot(ggg, aes(größe, gewicht))
> sd + geom_point(colour="red", size=3) +
+ scale_x_continuous("Größe",limits=c(150,190),breaks=seq(150,190,by=5),minor_breaks=NULL)
```

wird mit der Funktion **scale_x_continuous()** die Skala zur X-Achse modifiziert:

- Der Titel wird geändert.
- Die Endpunkte werden festgelegt.
- Per **seq()**-Funktion werden Haupteinteilungspunkte im 5er-Abstand gewählt.
- Die Nebenunterteilungspunkte (**minor_breaks**) werden abgeschaltet.

Offenbar kann man über den Operator **+** nicht nur Schichten zu einem Plot-Objekt hinzufügen, sondern auch andere Modifikationen vornehmen, z.B. eine Skala konfigurieren.

Ist ein Attribut mit einem diskreten Merkmal verbunden, werden den einzelnen Ausprägungen sukzessive die Darstellungsvarianten (z.B. Farben oder Symbole) aus einer vorgegebenen Palette zugeordnet. Über das **values**-Argument einer **scale**-Funktion kann man die Voreinstellungen durch individuelle Werte ersetzen, was im folgenden Beispiel mit den Füllfarben (Attribut **fill**) geschieht:

```
> gsd + scale_fill_manual(values=c("violet","lightcyan4"))
```

Eine Legende wird dann von **ggplot2** bei Bedarf automatisch eingefügt, wenn für eine visuelle Eigenschaft die Rückübersetzung in Datenwerte erläutert werden muss. Das ist z.B. erforderlich, wenn in einem gruppierten Streudiagramm Markierungen mit unterschiedlichen Farben auftreten:

```
> gsd <- ggplot(ggg, aes(größe, gewicht, colour=geschlecht)) + geom_point(size=3)
```

Bei der Legendenerstellung versucht **ggplot2** die Beiträge mehrerer Skalen (bzw. ästhetischer Attribute) zu kombinieren, so dass z.B. in einem gruppierten Streudiagramm mit gruppenspezifischen Anpassungsfunktionen eine Kombilegende mit Symbolfarbe und Linientyp entsteht, z.B.:



Ist der zu einem Geom erscheinende Legendenbeitrag unerwünscht, lässt er sich mit dem Argument **show_guide** der Geom-Funktion abschalten, z.B. beim folgenden Geom zur Anzeige von Beschriftungen für Datenpunkte in einem gruppierten Streudiagramm (vgl. Abschnitt 9.3.2):

```
> gsd <- ggplot(ggg, aes(größe, gewicht, colour=geschlecht)) + geom_point(size=3)
> gsd + geom_text(aes(y=gewicht-1, label=rownames(ggg)), show_guide=FALSE)
```

Im Beispiel wird so vermieden, dass aufgrund der Text-Geoms an die Legendensymbole ein unvorteilhaftes „a“ angeheftet wird:

Unerwünschter Legendenbeitrag	Korrekte Lösung
Geschlecht Frau Mann	Geschlecht Frau Mann

Um den Titel einer Achse oder Legende zu ändern, kann man die zum ästhetischen Attribut gehörige **scale**-Funktion aufrufen und das **name**-Attribut auf den gewünschten Wert setzen, z.B.:

```
> gsd + scale_colour_discrete(name="Geschlecht")
```

In Abschnitt 9.3.1.4.2 wird eine bequemere Alternative zur **scale()** – Funktion vorgestellt für den Fall, dass lediglich Titel zu ändern sind.

9.3.1.4.1 Wertebereiche

Geht es lediglich um eine Änderung der zu berücksichtigenden Wertebereiche geht, sind die folgenden „Bequemlichkeitsfunktionen“ (engl.: *convenience functions*) einfacher zu handhaben als die **scale**-Funktionen:

- **xlim()**
  Mit dieser Funktion legt man den Wertebereich der X-Achse fest, z.B.:
  ```
  > sd + xlim(150,190)
  ```
- **ylim()**
  Mit dieser Funktion legt man den Wertebereich der Y-Achse fest, z.B.:
  ```
  > sd + ylim(50,90)
  ```

Fälle außerhalb dieser Grenzen werden ausgeschlossen, was sich auch auf andere Schichten (z.B. mit einer Regressionsfunktion) auswirkt.

9.3.1.4.2 Beschriftungen

Geht es lediglich um eine Änderung der Beschriftung, sind die folgenden Funktionen einfacher zu handhaben als die **scale**-Funktionen:

- **xlab()**
  Mit dieser Funktion lässt sich die Beschriftung der X-Achse ändern, z.B.:
  ```
  xlab("Größe")
  > sd + xlab("Größe")
  ```
- **ylab()**
  Mit dieser Funktion lässt sich die Beschriftung der Y-Achse ändern, z.B.:
  ```
  > sd + ylab("Gewicht")
  ```
- **ggtitle()**
  Mit dieser Funktion kann man einen Diagrammtitel ergänzen, wobei ein Zeilenwechsel mit der Escape-Sequenz **\n** (New Line) veranlasst wird, z.B.:
  ```
  > sd + ggtitle("Regression von Gewicht\nauf Größe und Geschlecht\n")
  ```
- **labs()**
  Mit dieser Funktion kann man die Beschriftung von Achsen, Legende und Diagramm ändern, z.B.:
  ```
  > sd + labs(x="Größe", y="Gewicht", colour="Geschlecht")
  ```
  Eine Legendenbeschriftung muss dem zuständigen ästhetischen Attribut zugewiesen werden. Auch ein Diagrammtitel lässt sich vereinbaren, z.B.:
  ```
  > sd + labs(title="Regression von Gewicht\nauf Größe und Geschlecht\n")
  ```

Über **expression()** - Aufrufe können Formeln mit mathematischer Typographie in die Beschriftungen aufgenommen werden, z.B.

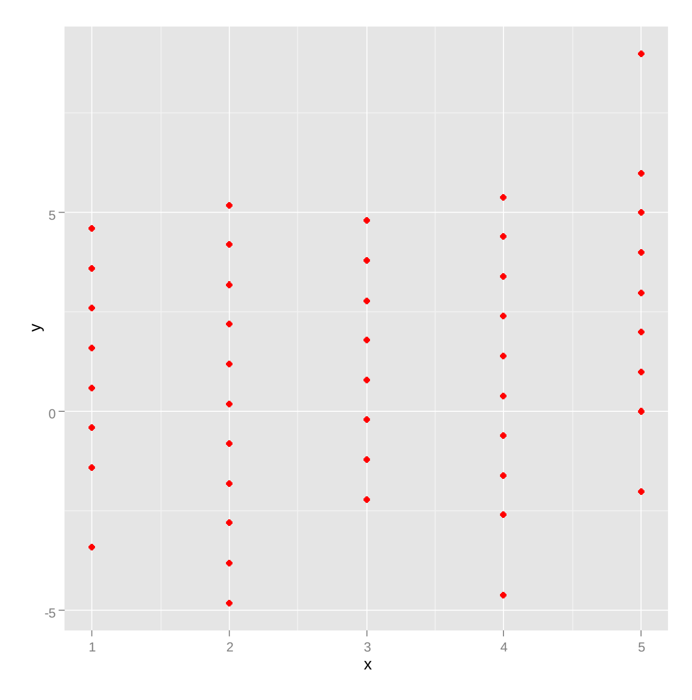
```
y=expression(f^2)
```

9.3.1.5 Positionsanpassungen

Um die Überlappung ihrer Elemente zu verhindern, kann eine Schicht **Positionsanpassungen** vornehmen (siehe Wickham 2009, Abschnitt 4.8). Bei einem Streudiagramm hilft manchmal die einfache Jitter-Technik, wobei kleine Zufallswerte zu den Daten addiert werden (vgl. Abschnitt 9.2.4.2.5). Im folgenden Beispiel mit simulierten Daten

```
> n <- 200
> set.seed(18)
> x <- round(runif(n, 1, 5), digits=0)
> e <- round(rnorm(n, 0, 2), digits=0)
> y <- 0.6*x + e
> df <- data.frame(x,y)
```

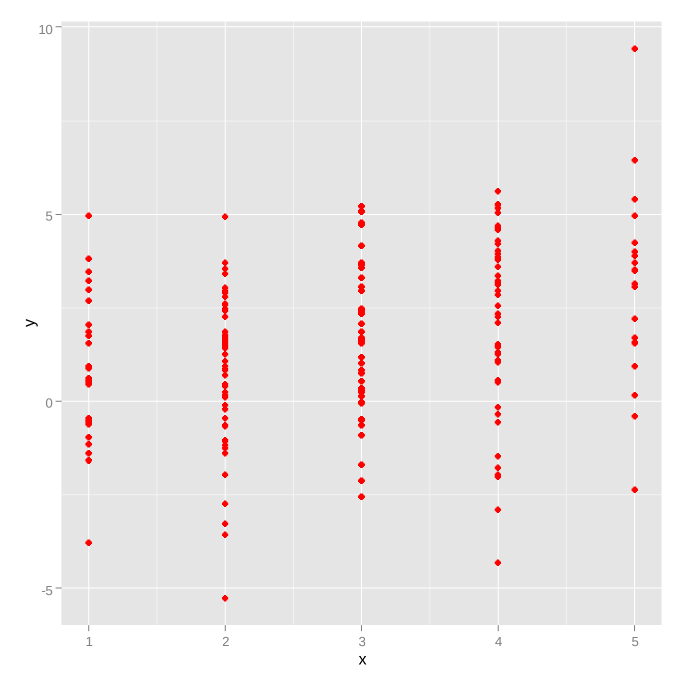
kann die Stärke der Beziehung von der Variablen x und y relativ schlecht beurteilt werden:



Daher wird über das **position**-Argument der Funktion **geom_point()** für eine Positionsanpassung gesorgt:

```
> ggplot(df, aes(x, y)) + geom_point(colour="red",
+  position = position_jitter(width=0, height=0.5))
```

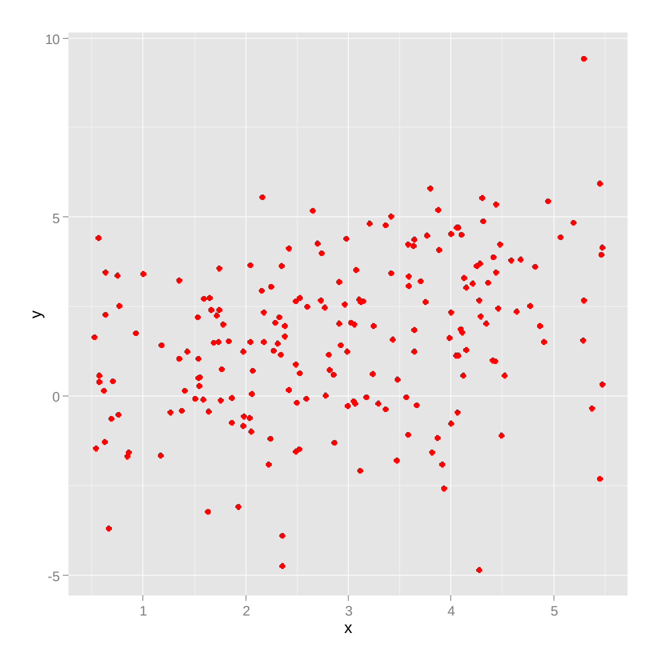
Es kommt die Funktion **position_jitter()** zum Einsatz, wobei die „Verwacklung“ in X- und Y-Richtung über entsprechende Argumente gesteuert werden kann. Im Beispiel bleiben die Regressorwerte unverändert, während die Kriteriumswerte eine Darstellungskorrektur erhalten. Nun sind die Dichteverhältnisse der bedingten Verteilungen und die Stärke der Beziehung besser zu beurteilen:



Werden auch die X-Positionen verwackelt,

```
> ggplot(df, aes(x, y)) + geom_point(colour="red",
+  position = position_jitter(width=0.5, height=0.5))
```

dann resultiert ein ausdrucksstarkes Diagramm, das allerdings erkennbar von der Realität mit 5-stufig erfasstem Regressor abweicht:



9.3.1.6 Facetten

Die in **ggplot2** realisierte *Grammar of Graphics* bietet die Möglichkeit der Facettierung, wobei die Gesamtstichprobe nach einer oder nach mehreren Variablen aufgespalten und für jede Teilstichprobe ein Diagramm (mit allen Schichten) erstellt wird.

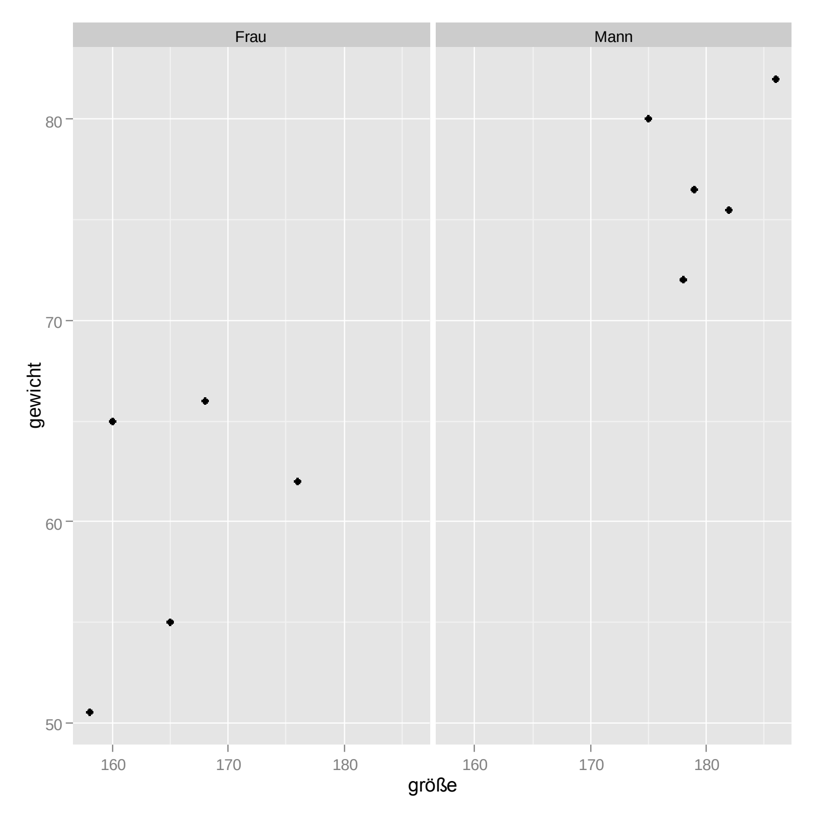
Um die Anordnung der einzelnen Diagramme zu kontrollieren, stehen zwei Facettierungs-Funktionen zur Verfügung:

- **facet_wrap()**
  Diese Funktion produziert eine Sequenz von Diagrammen, die per „Zeilenumbruch“ auf der rechteckigen Ausgabefläche angeordnet werden. Meist gibt man nur *eine* steuernde Variable an.
- **facet_grid()**
  Diese Funktion erlaubt es, durch *zwei* steuernde Variablen eine Anordnungsmatrix zu definieren.

Im folgenden Beispiel

```
> ggplot(ggg, aes(größe, gewicht)) + geom_point() + facet_wrap(~ geschlecht)
```

wird ein einfaches Streudiagramm (vgl. Abschnitt 9.3.2) mit den Variablen `größe` und `gewicht` definiert, wobei die Funktion **facet_wrap()** dafür sorgt, dass separate Diagramme für Frauen und Männer (die Stufen des Faktors `geschlecht`) erstellt werden (vgl. die einfache Variante in Abschnitt 9.3.1.1):



Um das Anordnungsdesign zu definieren, gibt man im Argument von **facet_wrap()** eine Tilde und anschließend die steuernde Variable an.

Ausführliche Informationen zur Facettierung sind im Kapitel 7 von Wickham (2009) zu finden.

9.3.1.7 Themes

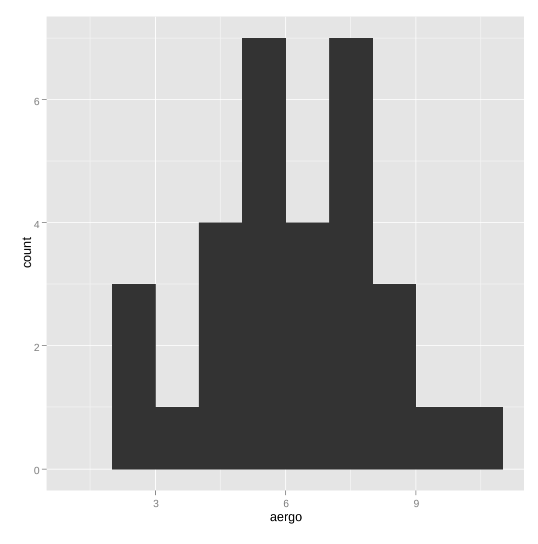
Über das Themes-System von **ggplot2** lassen sich Daten-unabhängige Aspekte im Erscheinungsbild eines Diagramms kontrollieren (z.B. Schriftarten und Farben von Texten). Man kann ...

- das Standard-Theme komplett durch eine Alternative (ein anderes Einstellungspaket) ersetzen
- und/oder einzelne Elemente eines Themes modifizieren

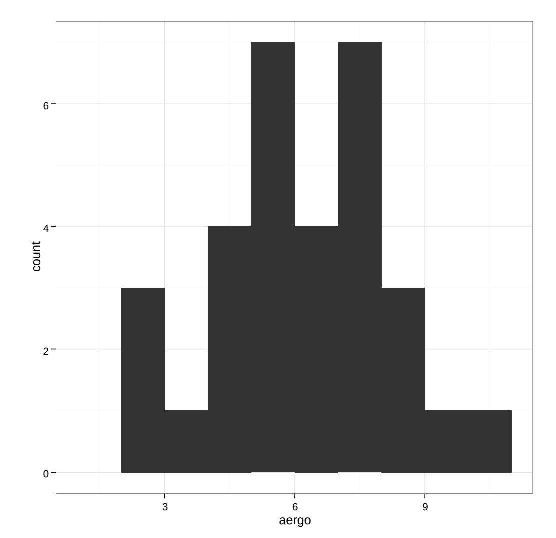
9.3.1.7.1 Themes im Ganzen

ggplot2 besitzt eingebaute Themen, die per Funktionsaufruf zu wählen sind. In der Version 1.0.0 sind vor allem zu nennen:

- **theme_grey()**
  Dieses Theme mit einem grauen Hintergrund und weißen Gitterlinien ist voreingestellt, z.B.:



- **theme_bw()**
  Dieses Theme mit einem weißen Hintergrund und grauen Gitterlinien liefert ein eher traditionelles Design, z.B.:



Über weitere Themes informiert die mit **?gglpot2** angeforderte Hilfeseite unter dem Stichwort **ggtheme**.

Um ein Theme auf ein einzelnes Diagramm anzuwenden, addiert man es zum Plot-Objekt, z.B.:

```
> histo + theme_bw()
```

Um ein Theme für *alle* im weiteren Verlauf einer **R**-Sitzung angefertigten **ggplot2**-Diagramme zu wählen, verwendet man die Funktion **theme_set()**, welche das bisherige Theme als Rückgabe liefert, so dass eine spätere Wiederherstellung möglich ist:

```
> actTheme <- theme_set(theme_bw())
```

Den Funktionen zur Theme-Auswahl ist gemeinsam, dass mit den Argumenten **base_size** bzw. **base_family** für die Beschriftungen eine Basisgröße und eine Schriftartenfamilie gewählt werden kann, z.B.:

```
> histo + theme_bw(base_size=14, base_family="serif")
```

In der Basisschriftgröße (Voreinstellung: 20pt) erscheinen die Achsentitel. Davon abgeleitete Größen sind:

- Größe des Titels: 120% der Basisgröße
- Größe der Teilstrichbeschriftungen: 80% der Basisgröße

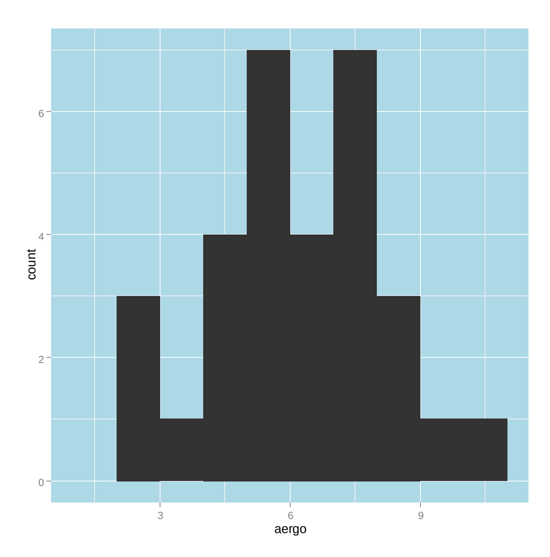
Zu den möglichen Schriftartenfamilien siehe Abschnitt 9.2.2.

9.3.1.7.2 Elemente von Themes modifizieren

Mit der Funktion **theme()**, welche die veraltete Funktion **opts()** ersetzt, kann man Elemente eines Themes modifizieren. Im folgenden Beispiel

```
> histo + theme(panel.background = element_rect(fill = "lightblue")
```

wird die Hintergrundfarbe eines Diagramms mit dem Standard-Theme verändert:



Auf der nach **?theme** erscheinenden Hilfeseite werden (in der **ggplot2**-Version 1.0.0) über 50 Elemente beschrieben, z.B.:

- **panel.background**
  Über die Elementfunktion **element_rect()** lässt sich der Hintergrund des Zeichenbereichs gestalten (z.B. Rand- und Füllfarbe über die Argumente **colour** und **fill**), z.B.:
  ```
  theme(panel.background = element_rect(fill = "lightblue")
  ```
- **axis.text.x, axis.text.y, plot.title**
  Über die Elementfunktion **element_text()** lassen sich Textattribute der Achsen- und Diagrammtitel beeinflussen (z.B. Schriftartenfamilie, Farbe und Größe über die Argumente **family**, **colour** und **size**), z.B.:
  ```
  theme(plot.title = element_text(colour = "red"))
  ```
- **axis.line**
  Über die Elementfunktion **element_line()** lassen sich Linienattribute von X- und Y-Achse gestalten, z.B. die Linienstärke über das Argument **size**:
  ```
  theme(axis.line = element_line(size = 2))
  ```

Über die Funktion **element_blank()** lässt sich die Ausgabe bestimmter Elemente unterdrücken, um z.B. ein Diagramm ohne Gitterlinien zu erstellen:

```
> histo + theme(panel.grid = element_blank())
```

Ein modifiziertes Theme lässt sich zur späteren Anwendung auf Diagramme in einem **R**-Objekt speichern, z.B.:

```
> redTitle <- theme_grey() + theme(plot.title = element_text(colour = "red"))
> histo + redTitle
```

9.3.2 Inkrementelle Erstellung eines gruppierten Streudiagramms

9.3.2.1 Plot-Objekt anlegen

In diesem Abschnitt wird die inkrementelle Diagrammerstellung mit **ggplot2** demonstriert. Starten Sie **R**, und laden Sie nötigenfalls das Paket ggplot2:

```
> library(ggplot2)
```

Als Beispiel erstellen wir ein gruppiertes Streudiagramm unter Verwendung der in Abschnitt 6.1 erstellten Beispieldaten mit den Variablen `geschlecht`, `größe` und `gewicht`:

```
> load("U:\\Eigene Dateien\\R\\ggg.RData")
> ggg
   geschlecht größe gewicht
1        Mann   186    82.0
2        Mann   178    72.0
3        Mann   182    75.5
4        Frau   160    65.0
5        Frau   168    66.0
6        Frau    NA    76.0
7        Frau   165    55.0
8        Mann   179    76.5
9        Frau   158    50.5
10       Mann   175    80.0
11       Frau   176    62.0
13       Mann   176      NA
```

Wir legen zunächst durch einen Aufruf der Funktion **ggplot()** ein Plot-Objekt an,

```
> sd <- ggplot(ggg, aes(größe, gewicht))
```

und vereinbaren dabei ...

- im ersten Argument die Datentabelle mit den zu visualisierenden Variablen
- im zweiten Argument per **aes()** - Aufruf eine Plot-globale Verknüpfung von ästhetischen Attributen mit Variablen (*aesthetic mapping*): Die Attribute **x** und **y** werden auf die Variablen `größe` und `gewicht` abgebildet, wobei die Namen der zugehörigen Funktionsargumente entfallen können, weil die Wertvergabe in Definitionsreihenfolge erfolgt (Positionsargumente).

Ein Plot-Objekt enthält keine Referenz auf die Datentabelle, sondern eine *Kopie*, so dass spätere Änderungen der Datentabelle ohne Effekt auf das Plot-Objekt bleiben.

9.3.2.2 Einfaches Streudiagramm

Bevor nicht mindestens eine Schicht erstellt worden ist, kann das Plot-Objekt noch nicht angezeigt werden. Daher machen wir uns daran, inkrementell die Schichten des Diagramms durch **geom**-Funktionen zu erzeugen und per + - Operator mit dem Plot-Objekt zu verknüpfen. Das folgende Kommando ergänzt eine Schicht mit den (x,y) - Datenpunkten und zeigt das Diagramm (durch einen impliziten **print()**-Aufruf) an:

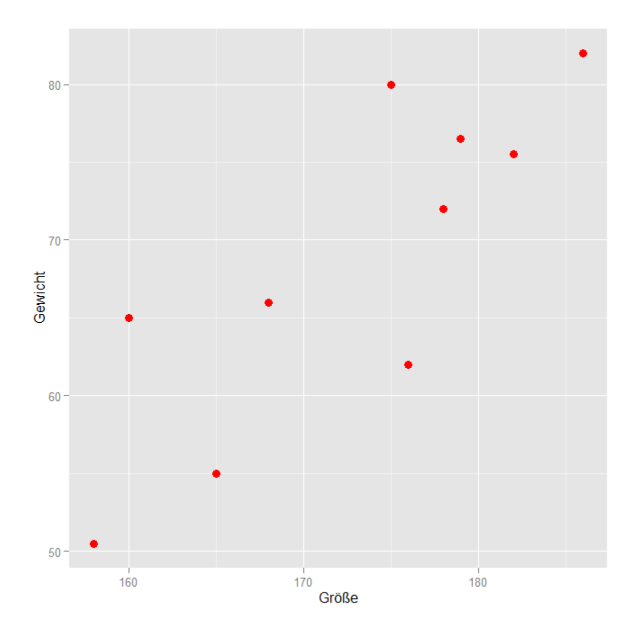
```
> sd + geom_point(colour="red", size=3)
```

Für die Punkte wird die Zeichenfarbe Rot gewählt und außerdem der Durchmesser erhöht, weil bei kleinen Kreisen (voreingestellte Form) die beim Windows-Ausgabegerät von **R** fehlende Kantenglättung unangenehm auffällt. Dabei wird jeweils ein ästhetisches Attribut auf einen festen Wert (statt auf eine Variable) abgebildet, und die Gültigkeit beschränkt sich auf die aktuelle Schicht.

Neben den **geom**-Funktionen, die jeweils eine neue Schicht anlegen, kennt das **ggplot2**-Paket Funktionen für Detailänderungen, die ebenfalls per + - Operator auf ein Plot-Objekt angewendet werden. Im folgenden Beispiel werden die Achsenbeschriftungen per **labs()** - Funktion modifiziert:

```
> sd + geom_point(colour="red", size=3) + labs(x = "Größe", y = "Gewicht")
```

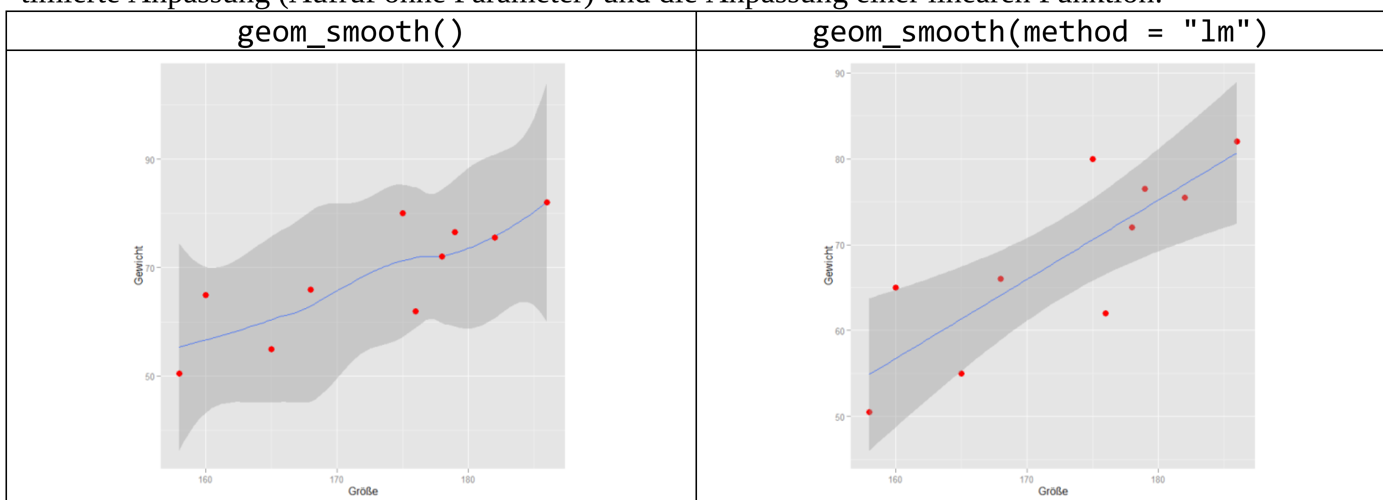
Das Ergebnis:



Dieses Diagramm ist der Bequemlichkeit halber aus dem Grafikfenster im RGui via Windows-Zwischenablage in das Manuskript übernommen worden, wobei eine bescheidene Auflösung von 96 dpi und somit eine suboptimale Qualität in Kauf genommen wurde. Über die Erstellung eines Diagramms in ansprechender Qualität informieren die Abschnitte 9.1.3 und 9.3.5.

9.3.2.3 Einfaches Streudiagramm mit Konfidenzzone

Um eine 95% - Konfidenzzone zu ergänzen, „addieren" wir eine Schicht mit einem Glättungs-Geom. Die zuständige Funktion **geom_smooth()** beherrscht unterschiedliche Glättungsverfahren, z.B. die lokal optimierte Anpassung (Aufruf ohne Parameter) und die Anpassung einer linearen Funktion:[1]



Damit die Markierungspunkte *über* der (partiell transparenten) Konfidenzzone liegen und nicht überlagert werden (siehe Negativbeispiel im rechten Diagramm), ist auf die richtige Reihenfolge der **geom**-Aufrufe zu achten. z.B.:

```
> sd + geom_smooth()+geom_point(colour="red",size=3)+labs(x="Größe",y="Gewicht")
```

[1] Zur Übernahme per Zwischenablage in Microsoft Word wurde das Bitmap-Format benutzt, weil beim Metafile-Transfer die Konfidenzzone verloren gegangen ist.

9.3.2.4 Gruppiertes Streudiagramm

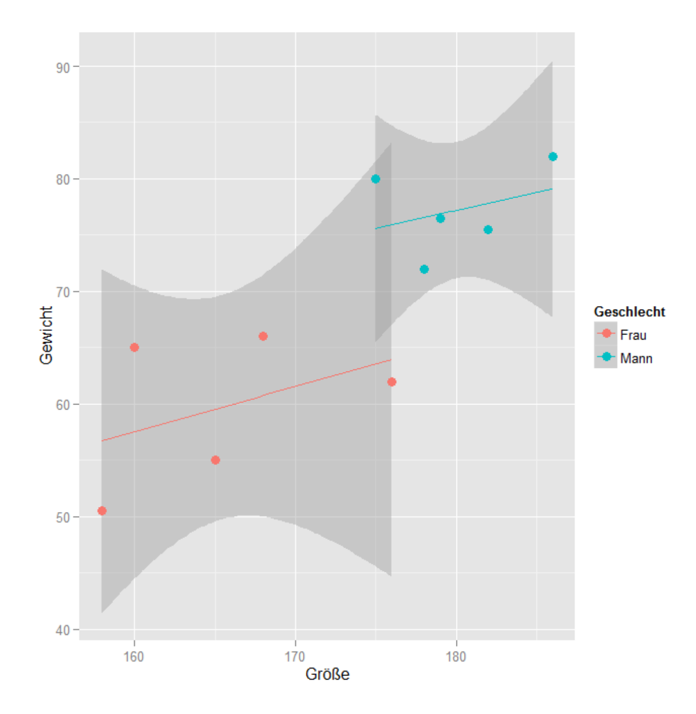
Um ein *gruppiertes* Streudiagramm zu erzielen, legen wir per **ggplot()**-Aufruf ein neues Plot-Objekt an und machen dabei die Markierungsfarbe von der Variablen `geschlecht` abhängig. Dazu wird im **aes()** -Aufruf das ästhetische Attribut **colour** an die Variable `geschlecht` gebunden:

```
> gsd <- ggplot(ggg, aes(größe, gewicht, colour=geschlecht))
```

Mit dem sukzessiven Diagrammaufbau

```
> gsd + geom_smooth(method="lm") + geom_point(size=3) +
+   labs(x="Größe", y="Gewicht", colour="Geschlecht")
```

erhalten wir das folgende Ergebnis, wobei auch die Anpassungsfunktion für das Glättungs-Geom auf die Gruppierung reagiert:



Um die Legendenbeschriftung zu ändern, wurde im **labs()**-Aufruf dem Argument (bzw. dem ästhetischen Attribut) **colour** die gewünschte Zeichenfolge zugewiesen.

Im aktuellen Fall erscheint es nicht sinnvoll, auf eine Legende zu verzichten. Trotzdem soll demonstriert werden, wie dies mit einer „additiv" ergänzten Theme-Modifikation (vgl. Abschnitt 9.3.1.7.2) möglich ist:

```
> gsd + geom_smooth(method="lm") + geom_point(size=3) +
+   labs(x="Größe", y="Gewicht") + theme(legend.position="none")
```

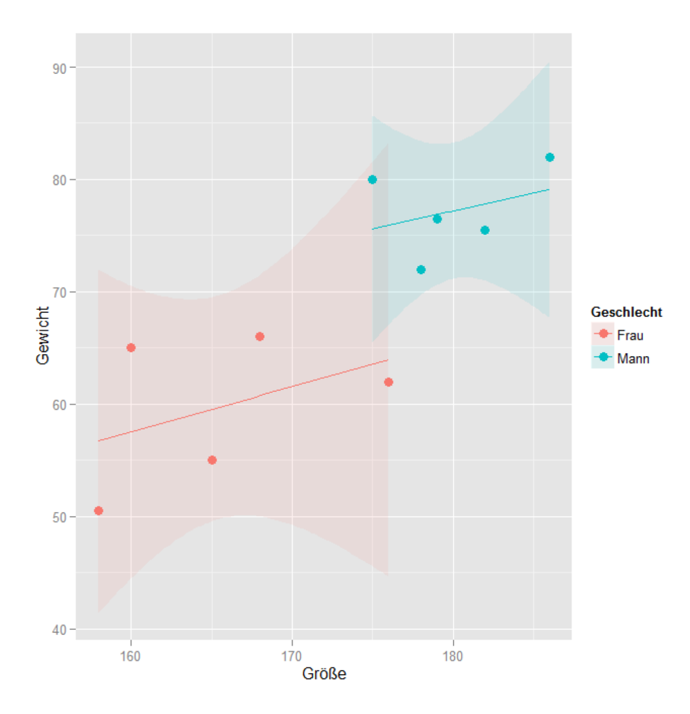
Wie man die Attribute einzelner Geome aus der Legende fernhält, wurde in Abschnitt 9.3.1.4 beschrieben. Im Beispiel kann man auch durch eine doppelte Verzichtserklärung für das komplette Verschwinden der Legende sorgen:

```
> gsd + geom_smooth(method="lm", show_guide=FALSE) +
+   geom_point(size=3, show_guide=FALSE) + labs(x="Größe", y="Gewicht")
```

Im folgenden Kommando nach einem Vorschlag von Field (2012, S. 141) werden die Konfidenzzonen Geschlechts-abhängig gefärbt und außerdem mit einem stärkeren Transparenzgrad versehen:

```
> gsd+geom_point(size=3)+geom_smooth(method="lm",aes(fill=geschlecht),alpha=0.1)+
+   labs(x="Größe", y="Gewicht")
```

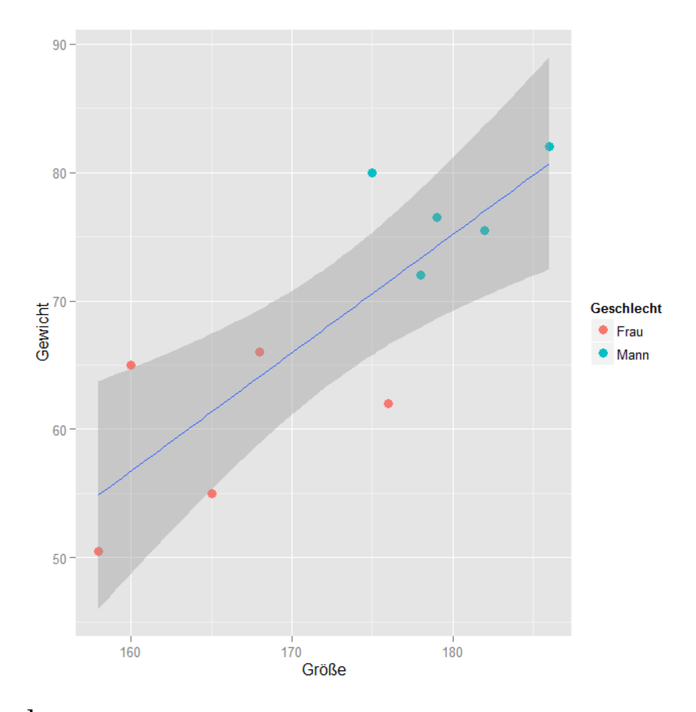
Ergebnis:



Wenn sich die gruppenspezifischen Anpassungsfunktionen kaum unterscheiden, kommt eine gemeinsame Anpassungsfunktion in Frage, wobei es aber in der Regel trotzdem sinnvoll ist, die Gruppen farblich zu unterscheiden. Sobald auf Plot-Ebene dem ästhetischen Attribut **colour** eine Variable zugewiesen ist, besteht eine Gruppierung, die sich auch auf das Glättungs-Geom auswirkt. Um dies zu verhindern, muss im **geom_smooth()** - Aufruf das ästhetische Attribut **group** auf den konstanten Wert 1 abgebildet werden:

```
> gsd + geom_smooth(method="lm",aes(group=1),show_guide=FALSE)+geom_point(size=3)+
+   labs(x="Größe", y="Gewicht", colour="Geschlecht")
```

Hier ist das gewünschte Ergebnis:



Um die unpassende Legende



durch eine Alternative ohne gruppenspezifische Linienfarben zu ersetzen, wird im **geom_smooth()** - Aufruf mit dem Argument **show_guide=FALSE** verhindert, dass die Schicht in der Legende Berücksichtigung findet.

9.3.2.5 Schichtaufbau mit qplot() starten

Am grundsätzlichen Schichtaufbau eines **ggplot2**-Diagramms ändert sich übrigens nichts, wenn statt **ggplot()** die Funktion **qplot()** verwendet wird. Ein **qplot()** - Aufruf erstellt ein Plot-Objekt und ergänzt Schichten. Man kann sogar eine **qplot()** - Produktion als Ausgangsbasis für den weiteren Schichtaufbau mit **geom**-Funktionen verwenden, z.B.:

```
> qplot(gewicht, größe, data = ggg) + geom_smooth(method="lm")
```

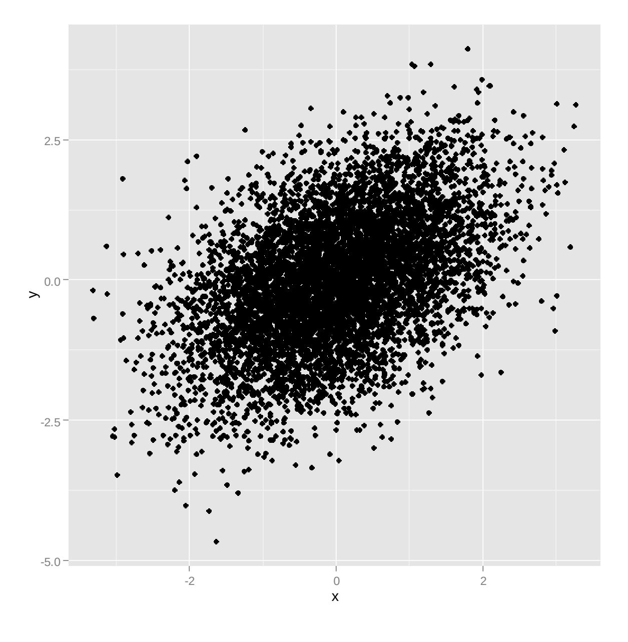
Im Beispiel landet allerdings die Glättungsschicht mit der transparenten Konfidenzzone *über* der Punkteschicht, was die Färbung einiger Punkte ändert. Die im Vergleich zu **qplot()** deutlich größere Flexibilität der Funktion **ggplot()** bei der Grafikproduktion zeigt sich u.a. in der perfekten Kontrolle über den Schichtaufbau.

9.3.2.6 Dichtedarstellung bei großen Stichproben

Wir haben schon in Abschnitt 9.2.4.2.6 im Zusammenhang mit der traditionellen **R**-Grafik Darstellungstechniken für Streudiagramme mit sehr vielen Fällen kennen gelernt. Wie das folgende Beispiel mit simulierten Daten

```
> n <- 7000
> set.seed(12)
> x <- rnorm(n,0,1)
> res = rnorm(n,0,1)
> y <- 0.5* x + res
> df <- data.frame(x,y)
> ggplot(df, aes(x,y)) + geom_point()
```

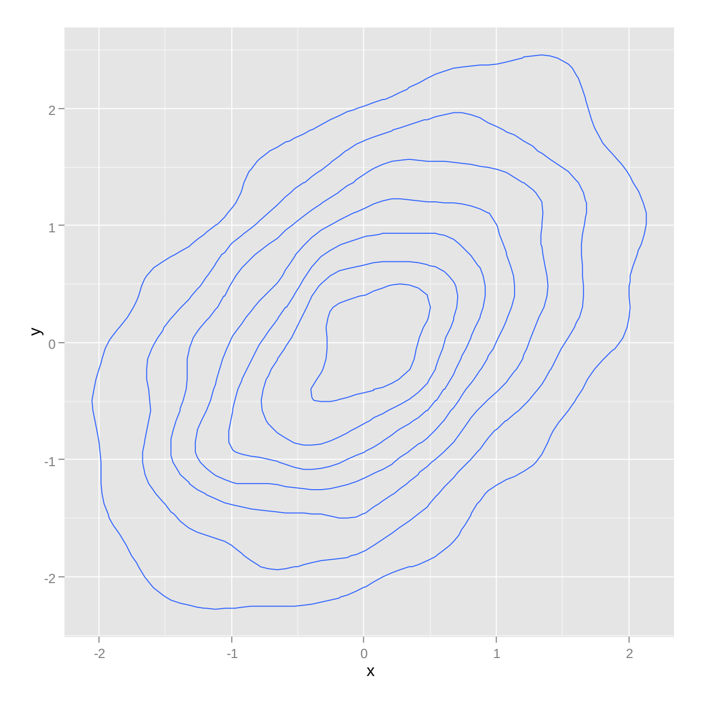
zeigt, ist die voreingestellte Darstellung von **geom_point()** nicht ideal:



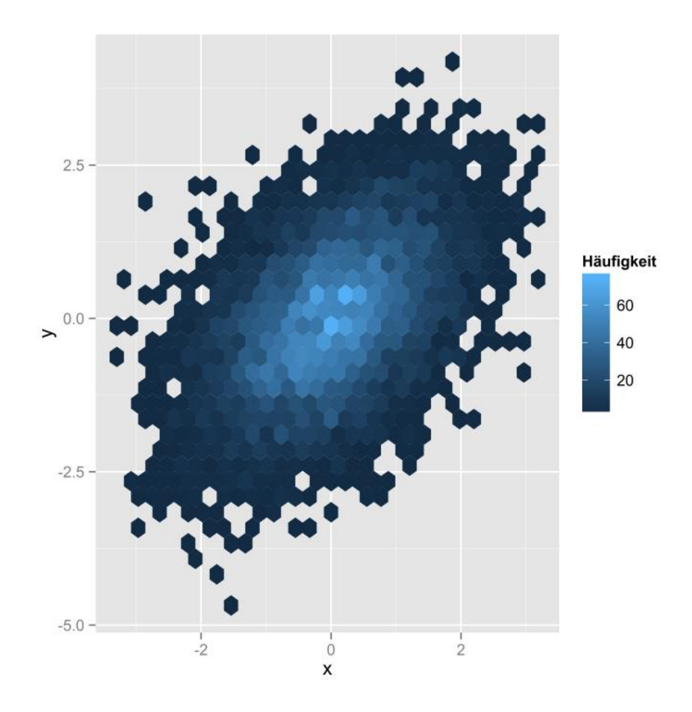
Von der Funktion **geom_density2d()**

```
> ggplot(df, aes(x,y)) + geom_density2d()
```

erhält man die **Dichtelinien** der Verteilung:



Eine in **ggplot2** (nach der Zusatzinstallation des Pakets **hexbin**) sowie auch in SPSS (via GPL) verfügbare Alternative ist das Streudiagramm mit **hexagonaler Gruppierung**. Dieses Exemplar



wurde durch folgende Syntax erzeugt:

```
> library(hexbin)
> ggplot(df,aes(x,y)) + stat_binhex() + scale_fill_continuous(name='Häufigkeit')
```

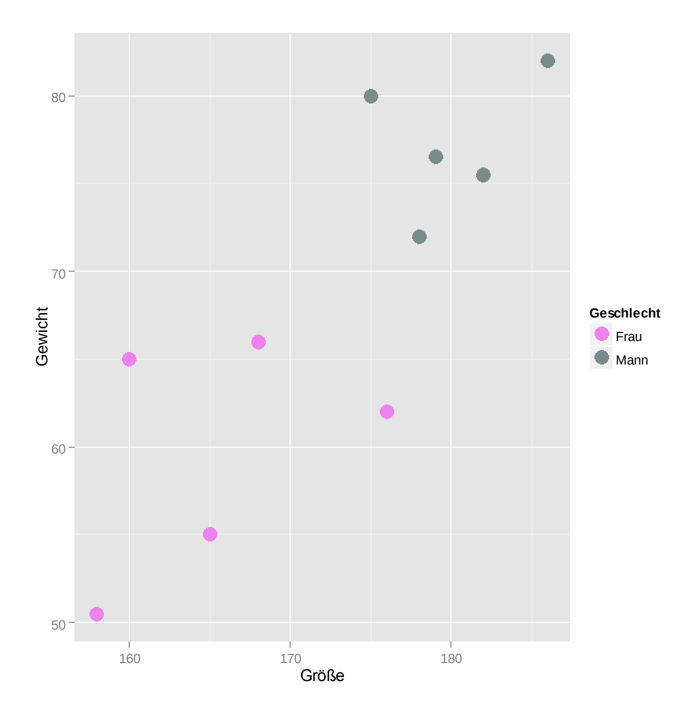
9.3.3 Werte für datengebundene ästhetische Attribute ändern

Die bei einem datengebundenen ästhetischen Attribut verwendeten Symbole, Linienstile und Farben lassen sich über eine **scale** - Funktion ändern. Zur Demonstration verwenden wir Varianten des bereits in Abschnitt 9.3.2 verwendeten gruppierten Streudiagramms.

Im ersten Beispiel werden für das zur Unterscheidung der Geschlechtsgruppen verwendete Attribut **colour** über das **values**-Argument der Funktion **scale_colour_manual()** individuelle Farben vereinbart:

```
> gsd <- ggplot(ggg,aes(größe,gewicht,colour=geschlecht))
> gsd <- gsd+geom_point(size=5)+labs(x="Größe",y="Gewicht")
> gsd + scale_colour_manual(name="Geschlecht", values=c("violet","lightcyan4"))
```

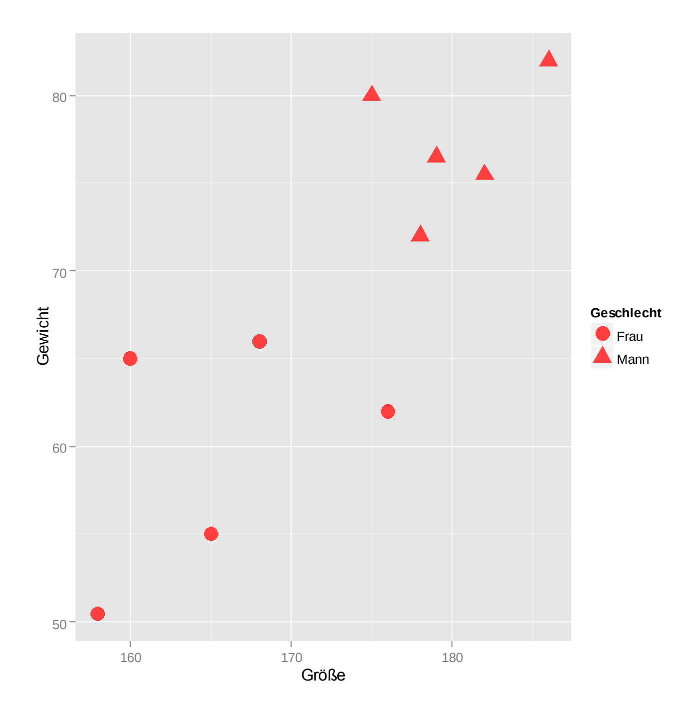
Dabei kommt das von **geom_point()** per Voreinstellung verwendete Symbol zum Einsatz (ein gefüllter Kreis):



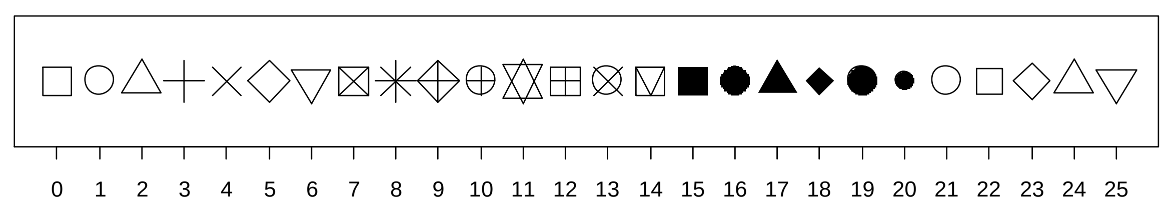
Dient das **shape**-Attribut zur Unterscheidung der Geschlechtsgruppen,

```
> gsd <- ggplot(ggg,aes(größe,gewicht,shape=geschlecht))
> gsd + geom_point(size=5,colour="brown1")+labs(x="Größe",y="Gewicht",shape="Geschlecht")
```

verwendet **geom_point()** per Voreinstellung die Symbole ● und ▲:



Dies sind die Symbole 16 und 17 aus der folgenden Palette, die sowohl von der traditionellen Grafik (vgl. Abschnitt 9.2.3.1) wie auch von **ggplot2** genutzt wird:



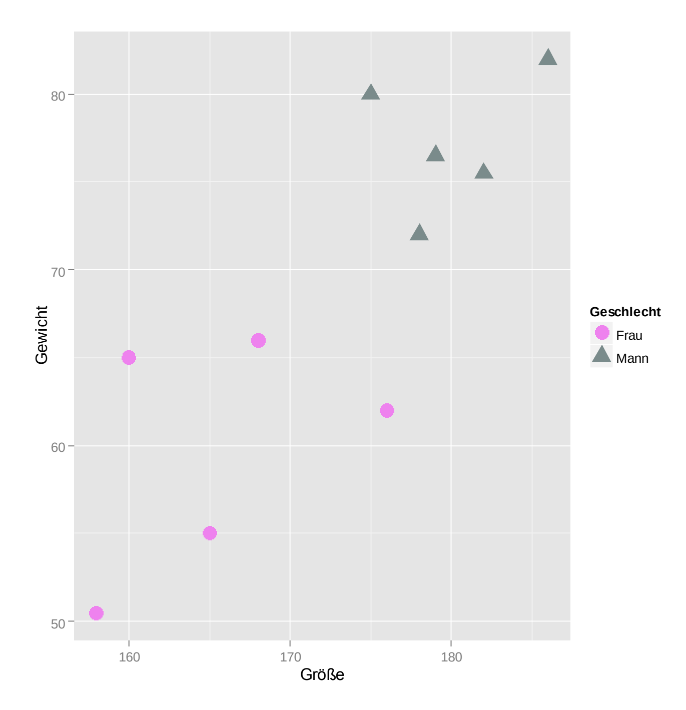
Über das **values**-Argument der Funktion **scale_shape_manual()** lassen sich alternative Symbole wählen:

```
> gsd <- ggplot(ggg,aes(größe,gewicht,shape=geschlecht))
> gsd <- gsd+geom_point(size=5,colour="brown1")+labs(x="Größe",y="Gewicht")
> gsd + scale_shape_manual(name="Geschlecht",values=c(15,18))
```

Selbstverständlich ist es möglich, Form und Farbe der Symbole simultan zur Gruppenunterscheidung zu verwenden:

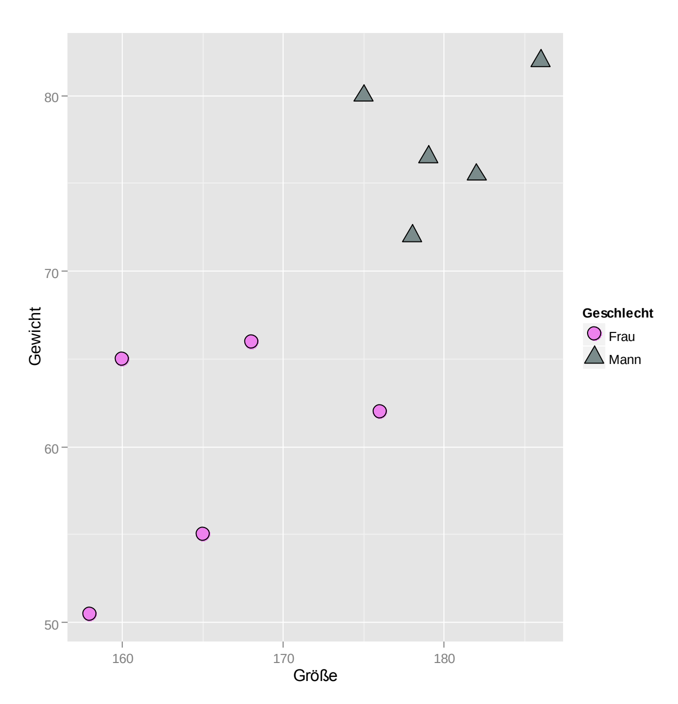
```
> gsd <- ggplot(ggg,aes(größe,gewicht,colour=geschlecht,shape=geschlecht))
> gsd <- gsd+geom_point(size=5)+labs(x="Größe",y="Gewicht",shape="Geschlecht")
> gsd + scale_colour_manual(name="Geschlecht",values=c("violet","lightcyan4"))
```

Im Beispiel werden die voreingestellten Symbole mit individuellen Farben verwendet:



Damit nur *eine* Legende erscheint, müssen unbedingt die beiden Legendentitel identisch gewählt werden.

Bei einigen Symbolen lassen sich Rahmen- und Füllfarbe getrennt ansprechen. Damit ist es z.B. möglich, eine datengebundene Füllfarbe mit einer datenunabhängigen Randfarbe zu kombinieren:



Im Beispiel werden die Symbole 21 und 24 mit individuellen Füllfarben für die Geschlechtsgruppen sowie der voreingestellten (datenunabhängigen) schwarze Randfarbe verwendet:

```
> gsd<-ggplot(ggg,aes(größe,gewicht,fill=geschlecht,shape=geschlecht))
> gsd<-gsd+geom_point(size=5)+labs(x="Größe",y="Gewicht",shape="Geschlecht",fill="Geschlecht")
> gsd+scale_fill_manual(values=c("violet","lightcyan4"))+scale_shape_manual(values=c(21,24))
```

9.3.4 Weitere Diagrammtypen

In diesem Abschnitt wird die Lösung wichtiger Visualisierungsaufgaben mit Hilfe des **ggplot2**-Pakets demonstriert.

9.3.4.1 Histogramm und Dichteplot

Zur Demonstration des Histogramms verwenden wir die SPSS-Datendatei **kfa.sav**, mit den Variablen aus dem statistischen Praktikum mit SPSS (siehe Baltes-Götz 2014c), die sich an dem im Vorwort vereinbarten Ort befindet. Die Variable `aergo` enthält den auf einer Skala von 0 bis 10 gemessenen Ärger von 31 Probanden über einen verpassten Flug.

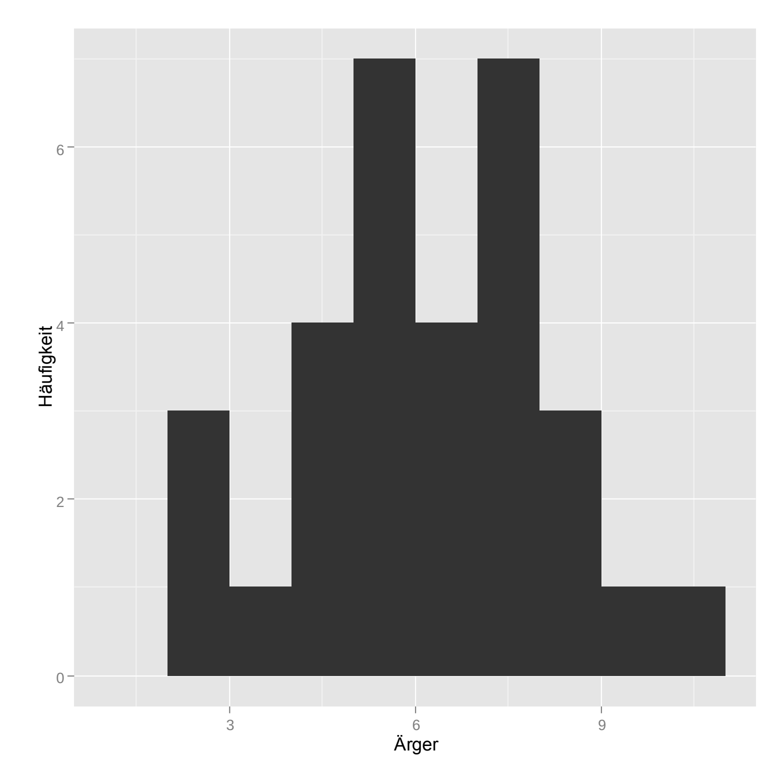
Zunächst erstellen wir mit der Funktion **ggplot()** ein Plot-Objekt und vereinbaren per **aes()** - Funktion die darzustellende Variable:

```
> histo <- ggplot(kfa, aes(aergo))
```

Wir ergänzen die Schicht mit dem Histogramm und sorgen im zuständigen **geom_histogram()** - Funktionsaufruf über das Argument **binwidth** für eine passende Intervallbreite. Außerdem werden die Achsenbeschriftungen per **labs()** - Funktion modifiziert:

```
> histo + geom_histogram(binwidth = 1) + labs(x="Ärger", y="Häufigkeit")
```

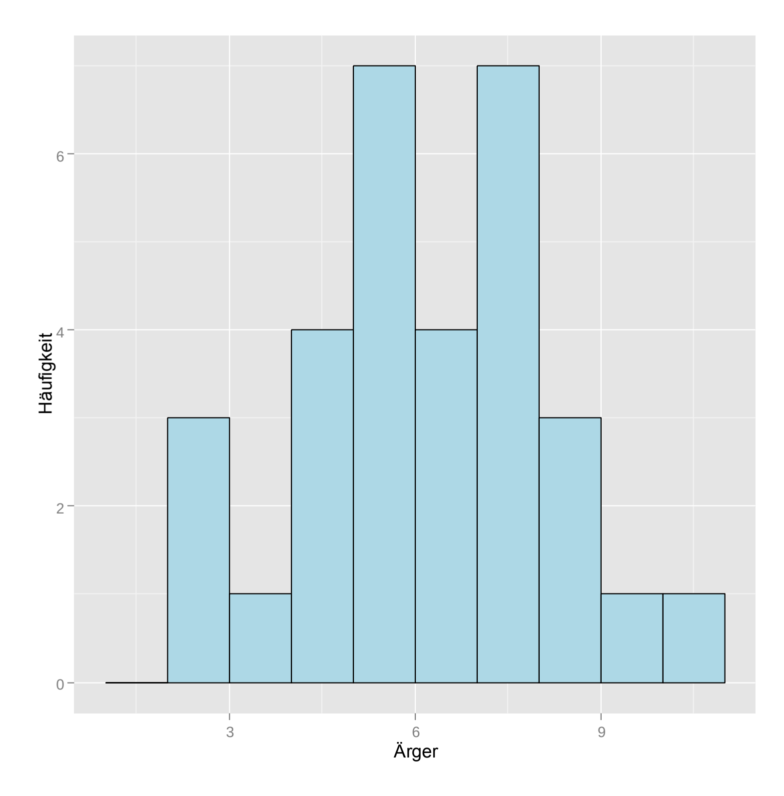
Im Ergebnis stört die triste schwarze Balkenfarbe:



Mit dem folgenden Kommando

```
> histo+geom_histogram(binwidth=1,colour="black",fill="lightblue")+labs(x="Ärger",y="Häufigkeit")
```

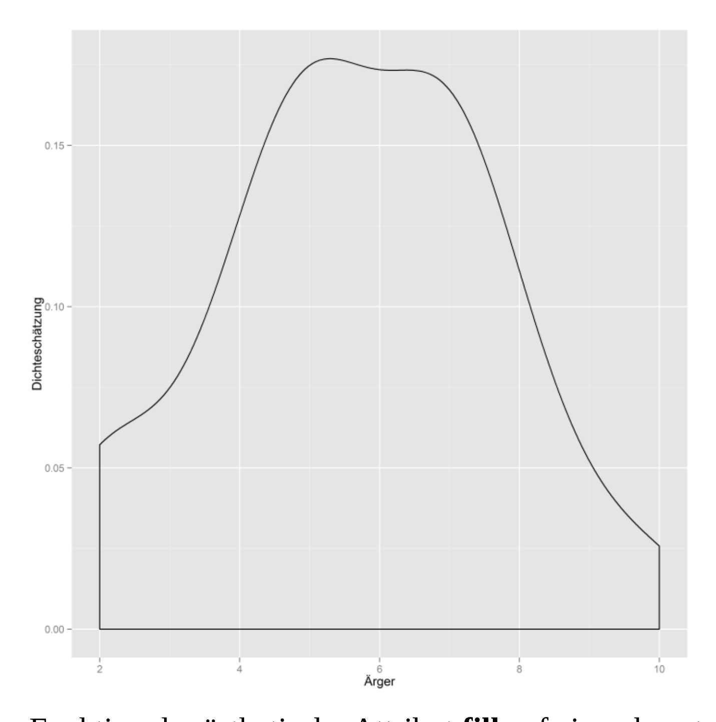
kommt eine freundlichere Farbe ins Spiel:



Über die Funktion **geom_density()** kann man die univariate Verteilung einer metrischen Variablen durch eine geschätzter Dichtefunktion beschreiben lassen.

```
> dichte <- ggplot(kfa, aes(aergo))
> dichte + geom_density() + labs(x="Ärger", y="Dichteschätzung")
```

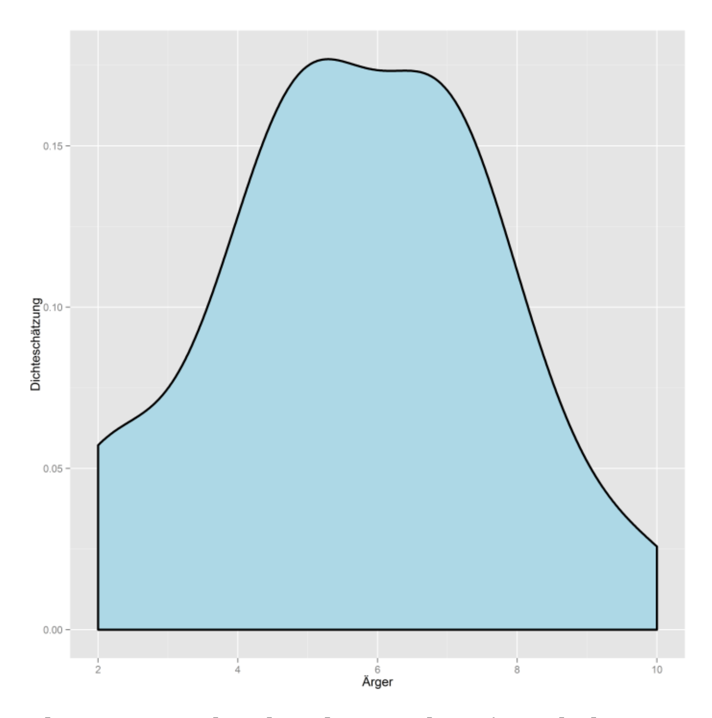
Im Beispiel resultiert eine „Mütze“:



Wird im Aufruf der Geom-Funktion das ästhetische Attribut **fill** auf einen konstanten Farbwert gesetzt,

```
> dichte + geom_density(fill="lightblue",size=1) + labs(x="Ärger", y="Dichteschätzung")
```

geht es etwas bunter zu:

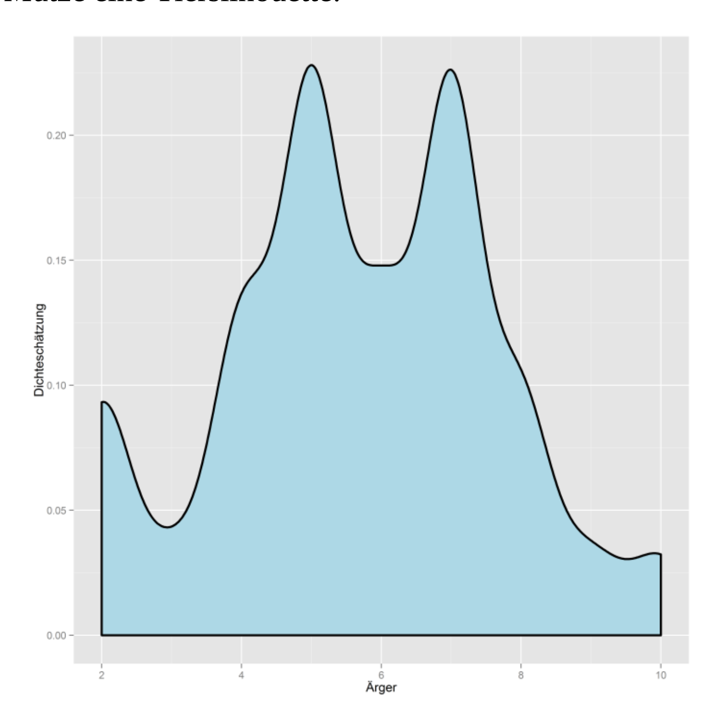


Im Beispiel wird außerdem die Linienstärke über das Attribut **size** erhöht.

Zur Steuerung des Glättungsgrads besitzt die Funktion **geom_density()** das Argument **adjust** mit dem Voreinstellungswert 1. Mit dem alternativen Wert 0,5

```
> dichte+geom_density(adjust=0.5,fill="lightblue",size=1)+labs(x="Ärger",y="Dichteschätzung")
```

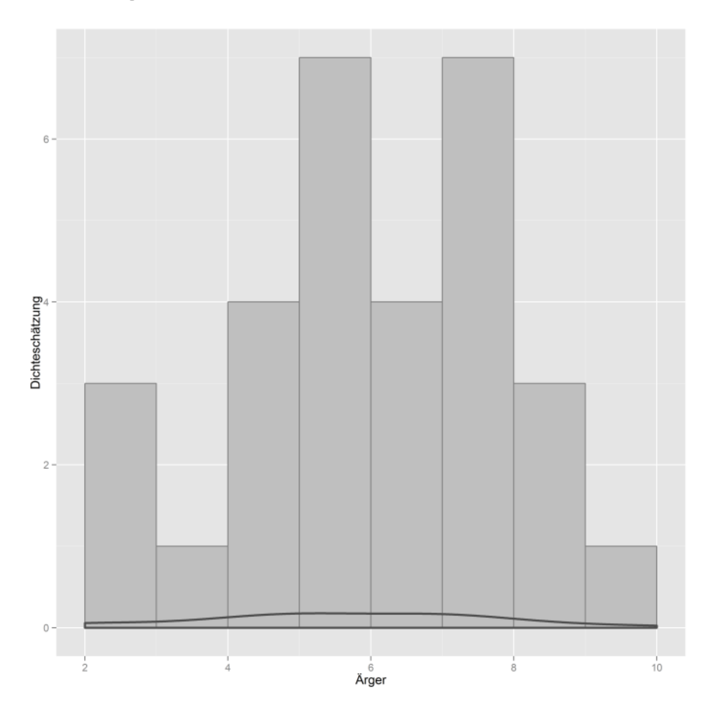
wird im Beispiel aus der Mütze eine Tiersilhouette:



Im folgenden Beispiel basierend auf einer Anregung aus Chang (2013, S. 125f) wird dem Histogramm eine Dichteschätzung überlagert. Von den Variablen, welche die bei **geom_histogram()** voreingestellte Transformation **stat_bin()** produziert, verwendet das Histogramm-Geom per Voreinstellung die Variable **count** mit den absoluten Häufigkeiten der Intervalle. Deren Werte übertreffen die Y-Werte von **geom_density()** (mit dem vorgeschriebenen Integrationsergebnis 1) bei weitem, so dass der erste Versuch

```
> histodichte <- ggplot(kfa, aes(aergo))
> histodichte + geom_histogram(binwidth=1,fill="gray75",colour="gray50") +
+   geom_density(size=1,colour="gray30")+labs(x="Ärger",y="Dichteschätzung")+xlim(2,10)
```

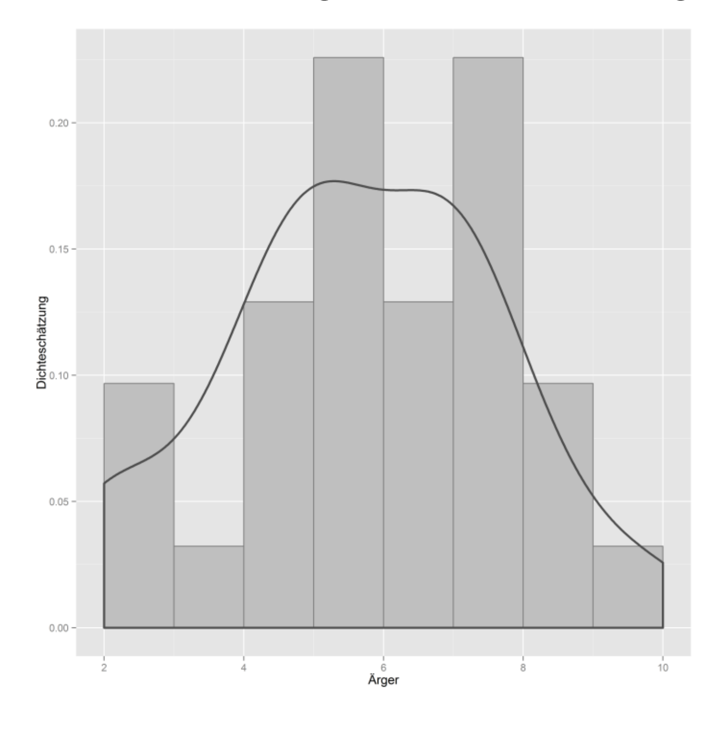
zu einem unbrauchbaren Gesamtergebnis führt:



Zur Lösung des Problems wird für das Histogramm-Geom die Y-Ache (ein ästhetisches Attribut) per **aes()** - Funktion auf die **stat_bin()** - Ergebnisvariable **density** abgebildet (zur Syntax vgl. Abschnitt 9.3.1.3):

```
> histodichte <- ggplot(kfa, aes(aergo))
> histodichte+geom_histogram(aes(y=..density..),binwidth=1,fill="gray75",colour="gray50")+
+   geom_density(size=1,colour="gray30")+labs(x="Ärger",y="Dichteschätzung")+xlim(2,10)
```

Durch Wahl passender Grauwerte kommen Histogramm und Dichtefunktion gut zur Geltung:



9.3.4.2 Boxplot

Zur Demonstration des Boxplots verwenden wir weiterhin die SPSS-Datendatei **kfa.sav** (siehe Abschnitt 9.3.4.1). Zunächst soll die Verteilung einer metrischen Variablen in der Gesamtstichprobe dargestellt werden. Wir erstellen mit der Funktion **ggplot()** ein Plot-Objekt und vereinbaren per **aes()** - Funktion die darzustellende Variable:

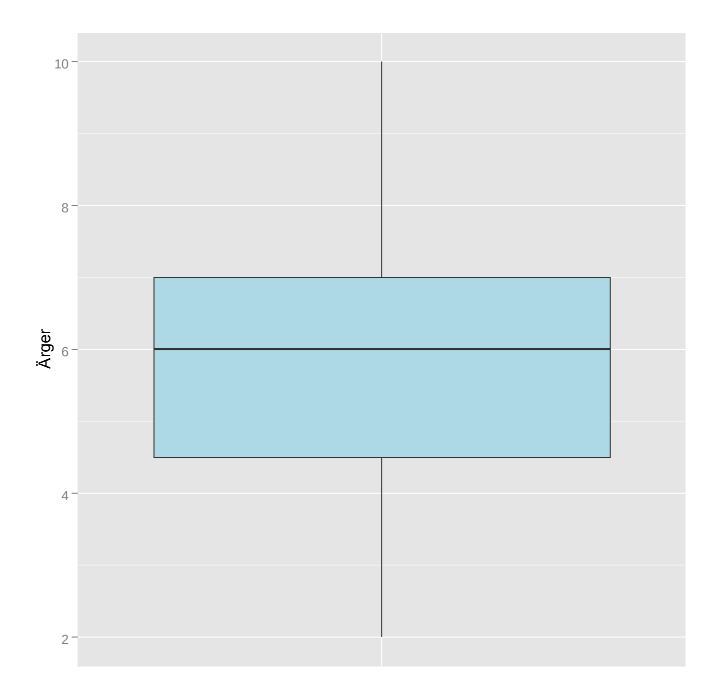
```
> box <- ggplot(kfa, aes(factor(0), aergo))
```

Weil das Boxplot-Geom eine Y-Achsenvariable mit der zu beschreibenden Verteilung und eine X-Achsenvariable zur Aufteilung der Stichprobe erwartet, liefern wir mit dem **factor(0)** einen Ersatz für die Gruppierungsvariable.

Wir ergänzen die Schicht mit dem Boxplot und sorgen im zuständigen **geom_boxplot()** - Funktionsaufruf über das Argument **colour** für eine angenehme Farbe. Mit der **labs()** - Funktion werden die Achsenbeschriftungen modifiziert, und der **theme()** - Aufruf unterdrückt die X-Achsen-Teilstrichbeschriftungen:[1]

```
> box + geom_boxplot(fill="lightblue") + labs(x="",y="Ärger") +
+   theme(axis.text.x = element_blank())
```

Das Ergebnis:



Sollen z.B. die Ärgerverteilungen bei Frauen und Männern gegenübergestellt werden, vereinbart man (z.B. bei der Erstellung des Plot-Objekts) eine passende Gruppierungsvariable, z.B.:

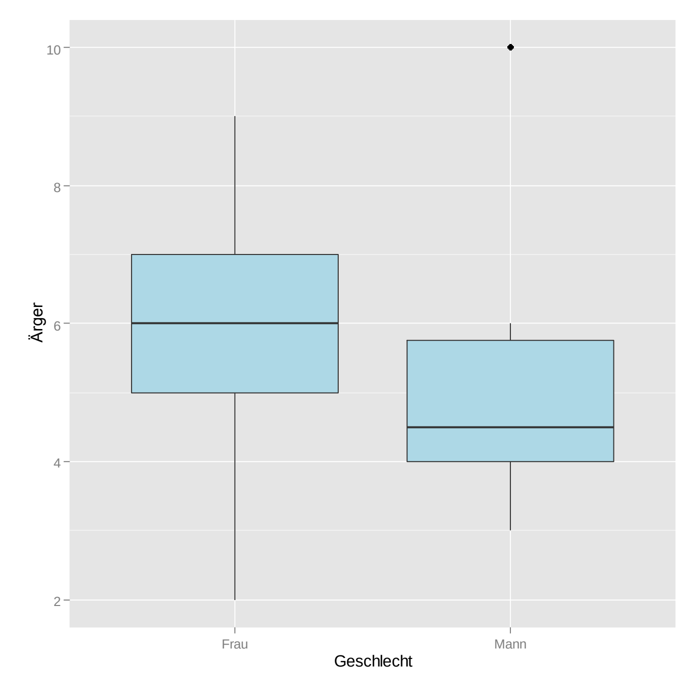
```
> box <- ggplot(kfa, aes(geschl, aergo))
```

Die Anweisung

```
> box + geom_boxplot(fill="lightblue") + labs(x="Geschlecht",y="Ärger")
```

liefert nun zwei nebeneinander stehende Boxplots, die einen Vergleich der beiden Verteilungen hinsichtlich Lage und Dispersion erlauben:

[1] Anderenfalls würde eine unmotivierte 0 als Teilstrichbeschriftung erscheinen.



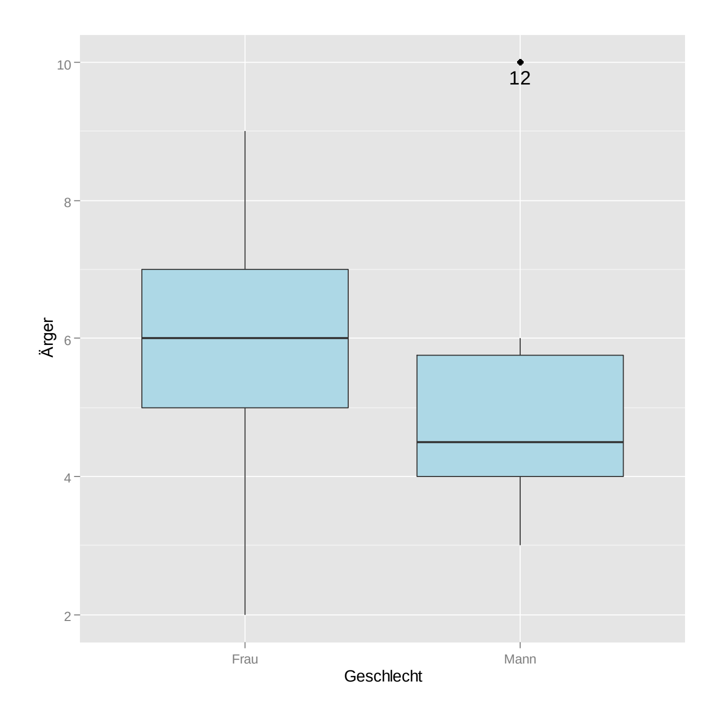
In der männlichen Teilstichprobe zeigt sich zudem ein Ausreißer, der um mehr als 1,5 Boxbreiten vom 3. Quartil (dem oberen Rand der Box) entfernt ist. Um diesen Fall zu etikettieren, wird eine Variable vorbereitet, welche bei Personen mit dem Ärgerwert 10 die Fallnummer (den Wert der Variablen `fnr`) enthält und ansonsten den Indikator für fehlende Werte:

```
> etikett <- kfa$fnr; etikett[kfa$aergo<10] <- NA
```

Nun wird eine Schicht mit **text**-Geom unter Verwendung der vorbereiteten Etikettierungsvariablen ergänzt:

```
> box + geom_boxplot(fill="lightblue") + labs(x="Geschlecht", y="Ärger") +
+   geom_text(aes(label=etikett), vjust=1.4)
```

Mit der Eigenschaft **vjust** wird der Text in vertikaler Richtung vom Datenpunkt weg bewegt, um eine Überlagerung zu verhindern:



9.3.4.3 Balkendiagramme

Mit einem einfachen Balkendiagramm kann man darstellen:

- der Verteilung einer diskreten (z.B. kategorialen) Variablen,
- die Mittelwerte oder andere statistische Zusammenfassungen einer metrischen Variablen für die Ausprägungen einer kategorialen Variablen,
- die Mittelwerte von mehreren metrischen Variablen.

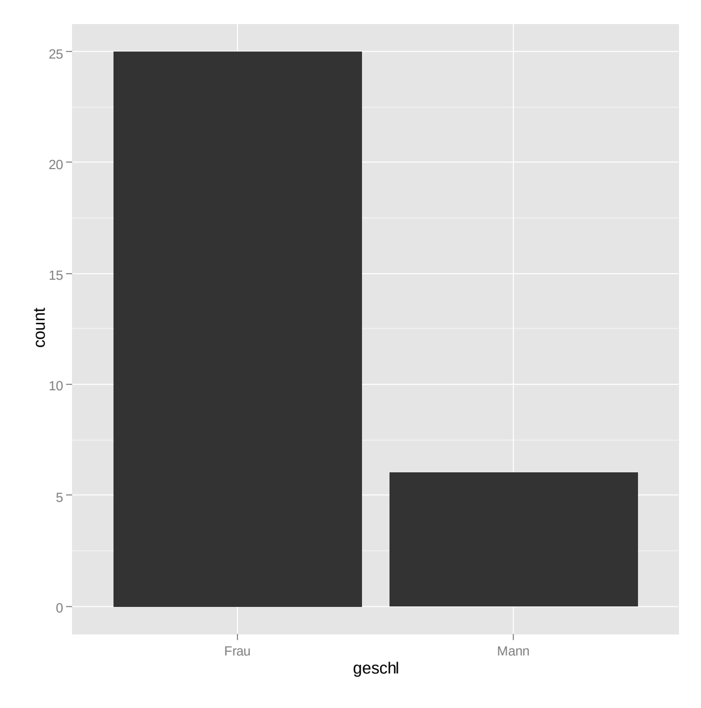
Zur Demonstration der verschiedenen Balkendiagramme verwenden wir weiterhin die SPSS-Datendatei **kfa.sav** (siehe Abschnitt 9.3.4.1).

9.3.4.3.1 Diskrete Verteilungen

Über die Funktion **geom_bar()** lässt sich mit sehr wenig Aufwand

```
> ggplot(kfa, aes(x=geschl)) + geom_bar()
```

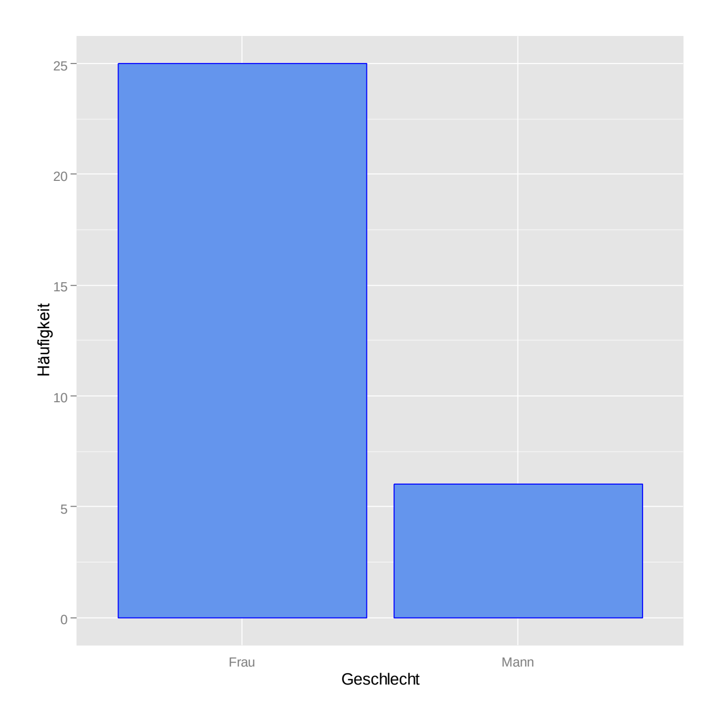
ein Balkendiagramm zur Anzeige der absoluten Häufigkeiten für die Kategorien einer diskret verteilten Variablen erstellen, z.B.:



Ein kleiner Zusatzaufwand

```
> ggplot(kfa, aes(x=geschl)) + geom_bar(fill="cornflowerblue", colour="blue") +
+   labs(x="Geschlecht", y="Häufigkeit")
```

erlaubt die wahlfreie Färbung und Achsenbeschriftung, z.B.:



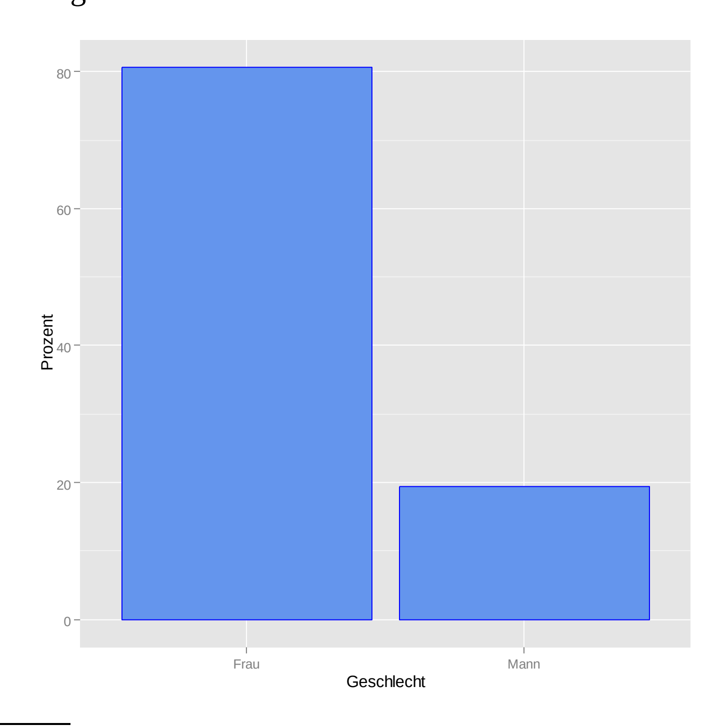
Indem das ästhetische Attribut **y** auf die folgende Funktion[1]

```
(..count..)*100/sum(..count..)
```

der Transformations-Ergebnisvariablen **..count..** abgebildet wird,

```
> ggplot(kfa,aes(x=geschl,y=..count..*100/sum(..count..))) +
+   geom_bar(fill="cornflowerblue",colour="blue") +
+   labs(x="Geschlecht",y="Prozent")
```

erhält man die relativen Häufigkeiten in Prozent:



[1] Eigentlich sollte das Ziel mit der Transformationsvariablen **density** leichter erreichbar sein. Es hat sich aber ein leicht seltsames Verhalten gezeigt. Wenn man das ästhetische Attribut **group** auf den Wert 1 setzt, klappt es mit der Variablen **density** (Tipp von http://stackoverflow.com/questions/17406082/using-density-in-stat-bin-with-factor-variables):

```
> ggplot(kfa,aes(x=geschl)) + geom_bar(aes(y=..density..*100, group=1), fill="cornflowerblue",colour="blue") +
+   labs(x="Geschlecht",y="Prozent")
```

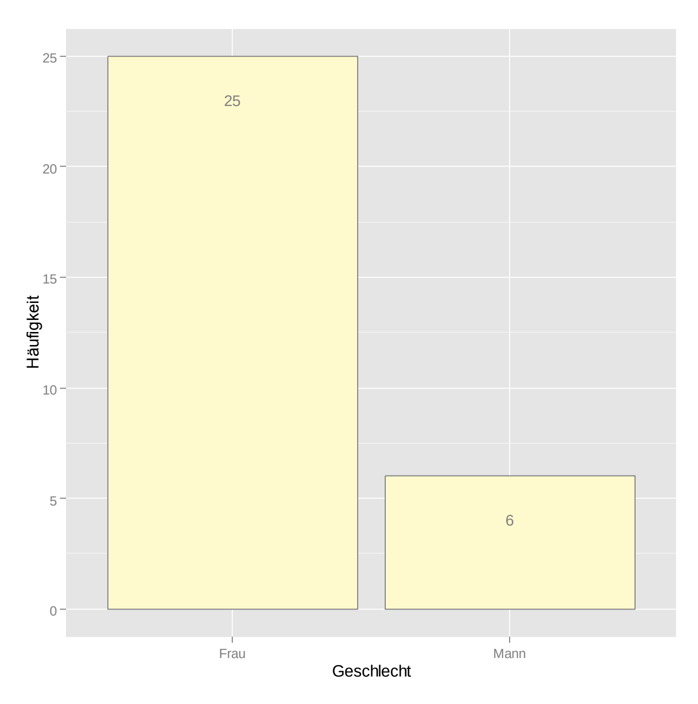
Sollen die Balken mit den absoluten Häufigkeiten beschriftet werden, kommt das **text**-Geom zum Einsatz:

```
> ggplot(kfa,aes(x=geschl)) + geom_bar(fill="lemonchiffon",colour="gray50") +
+   labs(x="Geschlecht",y="Häufigkeit") +
+   geom_text(stat="bin",mapping=aes(label=..count..),vjust=4,size=4,colour="gray50")
```

Die bei **geom_text()** voreingestellte identische Transformation wird durch **bin** ersetzt. Die somit verfügbare Variable **..count..** wird mit dem ästhetischen Attribut **label** verbunden, das die auszugebenden Texte festlegt. Andere ästhetische Attribute erhalten einen festen Wert:

- **vjust** legt eine vertikale Verschiebung der Texte relativ zu den (durch **..count..** festgelegten) **y**-Positionen fest.
- **size** gibt die Textgröße an.
- **colour** bestimmt die Textfarbe.

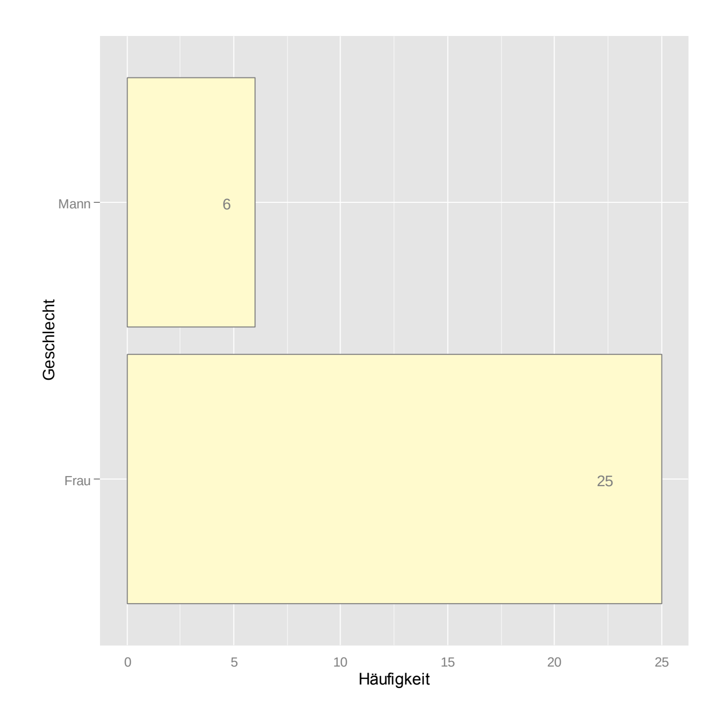
Auf der Schicht mit dem **text**-Geom erscheinen die gewünschten Etiketten mit den gewählten Attributen:



Um **waagerechte Balken** zu erhalten, muss man lediglich auf ein fertiges Balkendiagramm die Funktion **coord_flip()** anwenden. Im aktuellen Beispiel ist es allerdings ratsam, zusätzlich die Textpositionen neu zu adjustieren (**hjust** statt **vjust**):

```
> ggplot(kfa,aes(x=geschl)) + geom_bar(fill="lemonchiffon",colour="gray50") +
+   labs(x="Geschlecht",y="Häufigkeit") +
+   geom_text(stat="bin",mapping=aes(label=..count..),hjust=4,size=4,colour="gray50") +
+   coord_flip()
```

Das Ergebnis:

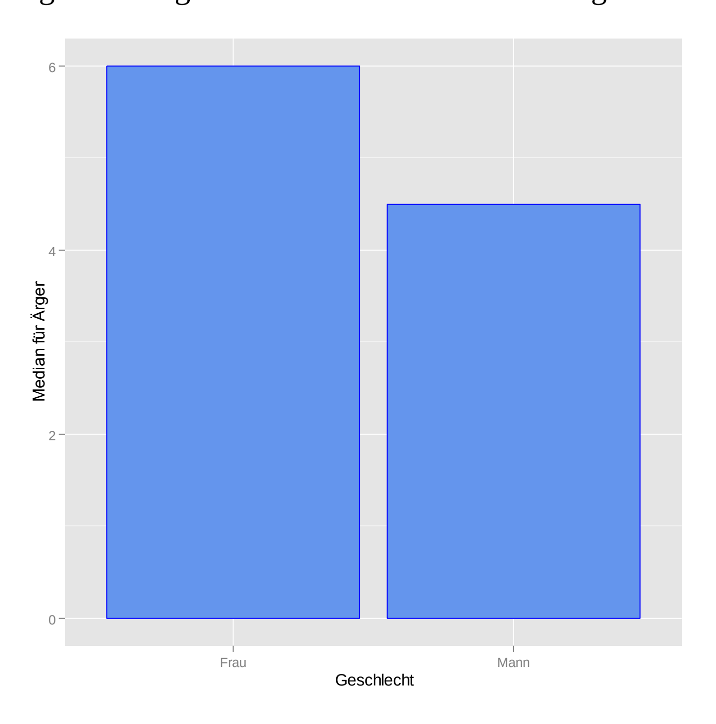


9.3.4.3.2 Statistische Kennwerte einer abhängigen Variablen für die Stufen eines Faktors

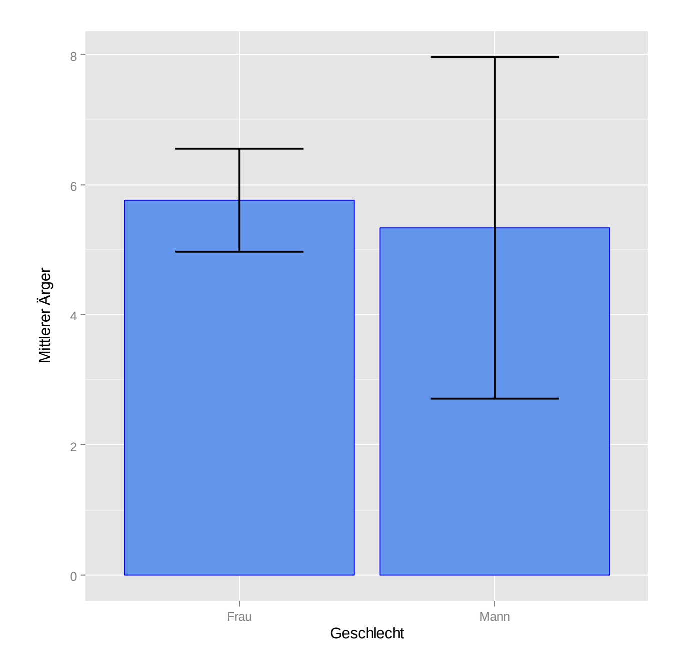
Kombiniert man **geom_bar()** mit der statistischen Transformation **stat_summary()** (an Stelle der Voreinstellung **stat_bin()**), kann man statistische Kennwerte einer abhängigen Variablen (z.B. Mittelwert Median, Maximum) für die Faktorstufen darstellen. Das ästhetische Attribut **y** lässt man von einer individuellen Zusammenfassungsfunktion liefern, die über das Argument **fun.y** bestimmt wird, z.B.:

```
> ggplot(kfa, aes(x=geschl, y=aergo)) +
+   geom_bar(stat="summary", fun.y=median, fill="cornflowerblue") +
+   labs(x="Geschlecht", y="Median für Ärger")
```

Das resultierende Balkendiagramm zeigt die Mediane der Variablen Ärger für Frauen und Männer:



Im nächsten Diagramm ersetzen wir den Median durch das arithmetische Mittel und ergänzen eine von der Funktion **geom_errorbar()** erstellte Schicht mit den normalverteilungsbasierten 95% - Vertrauensintervallen der beiden Gruppen:

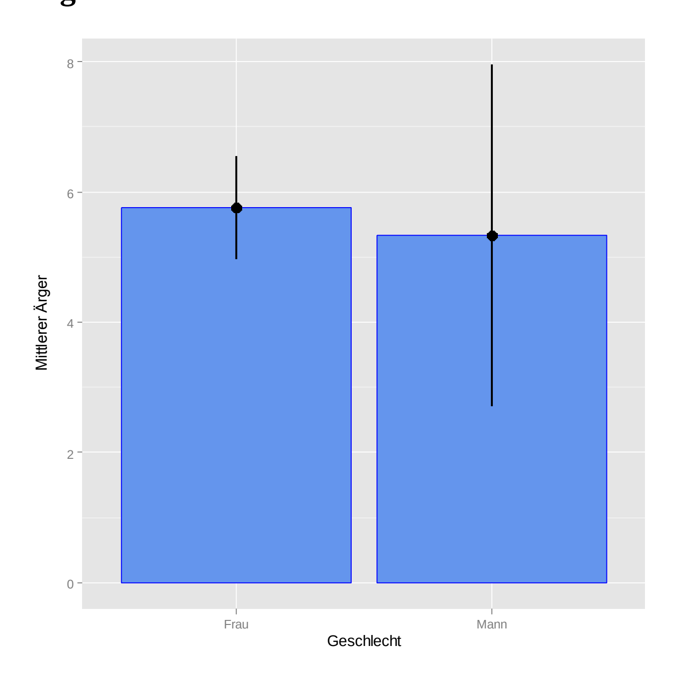


Auch in der Funktion **geom_errorbar()** kommt die **summary**-Transformation zum Einsatz:

```
> ggplot(kfa, aes(x=geschl, y=aergo)) +
+   geom_bar(stat="summary", fun.y=mean, fill="cornflowerblue") +
+   geom_errorbar(stat="summary", fun.data=mean_cl_normal, width=0.5, size=1) +
+   labs(x="Geschlecht", y="Mittlerer Ärger")
```

Dem Argument **fun.data** wird die Zusammenfassungsfunktion **mean_cl_normal()**, zugewiesen, deren Vektor-wertige Rückgabe die Vertrauensschranken enthält, welche die ästhetischen Attribute **ymin** und **ymax** des **errorbar**-Geoms mit Werten versorgen. Über die **errorbar**-Attribute **width** und **size** beeinflusst man die Breite und die Linienstärke der Dispersionsindikatoren.

Um Dispersionsindikatoren ohne horizontale Begrenzungen zu erzeugen, ersetzt man das Geom **errorbar** durch die Alternative **pointrange**:



9.3.4.3.3 Statistische Kennwerte von mehreren Variablen

Während ein Balkendiagramm zur Darstellung der Effekte eines Gruppierungsfaktors in **ggplot2** leicht zu erstellen ist, muss man bei einem Messwiederholungsfaktor etwas mehr Aufwand investieren. In der Beispieldatei **kfa.sav** sind die Ärgereinschätzungen der Probanden für zwei Bedingungen enthalten: Bei Ab- bzw. Anwesenheit einer als *KFA* bezeichneten Konstellation (Variablen `aergo` bzw. `aergm`).[1] Weil **ggplot2** für ein Balkendiagramm mit den beiden Variablen einen Faktor für die X-Achse benötigt, konvertieren wir die Daten mit Hilfe der Funktion **melt()** aus dem Paket **reshape2** in das Langformat, wobei ein Faktor namens `kfa` mit den Stufen `aergo` und `aergm` entsteht:

```
> library(reshape2)
> kfa.long <- melt(kfa, id.vars = "fnr",
+   measure.vars = c("aergo","aergm"), variable.name = "kfa")
> kfa.long$kfa <- factor(kfa.long$kfa, labels=c("Ohne","Mit"))
```

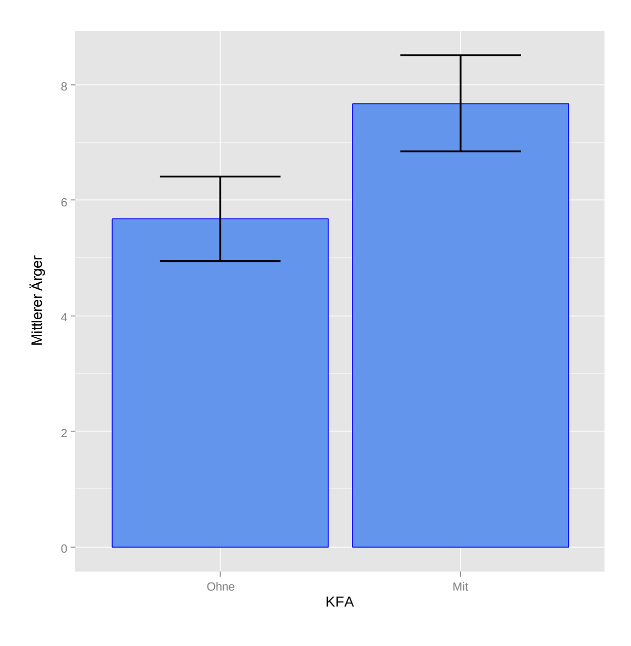
Wir erhalten die folgende Datentabelle `kfa.long` mit dem Faktor `kfa` und dem numerischen Vektor `value`:

```
   fnr  kfa value
1    1 Ohne     5
2    2 Ohne     5
 . . .
31  31 Ohne     7
32   1  Mit     8
33   2  Mit     8
 . . .
62  31  Mit     9
```

Um dazu per ein Balkendiagramm mit Fehlerbalken zu erstellen, kann die **ggplot2**-Syntax aus Abschnitt 9.3.4.3.2 weitgehend übernommen werden:

```
> ggplot(kfa.long,aes(x=kfa,y=value)) +
+   geom_bar(stat="summary", fun.y=mean, fill="cornflowerblue", colour="blue") +
+   geom_errorbar(stat="summary", fun.data=mean_cl_normal, width=0.5, size=1) +
+   labs(x="KFA", y="Mittlerer Ärger\n")
```

Der nicht unerhebliche Aufwand wird durch ein gutes Ergebnis belohnt:



[1] Bei der im aktuellen Kontext wenig relevanten Konstellation geht es um die kontrafaktische (also nicht eingetretene) Alternative zu einem ungünstigen Ereignis.

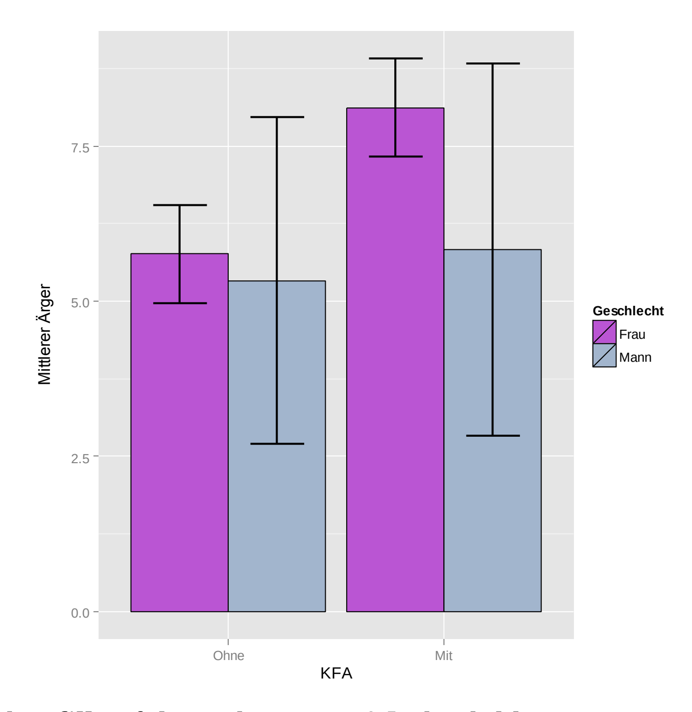
Mit SPSS ist ein weitgehend identisches Diagramm mit wenigen Mausklicks zu erstellen. Allerdings bietet die Verwendung von **R** mehr Flexibilität, die z.B. zur Erstellung von korrigierten Fehlerbalken genutzt werden könnte: Bei der beschriebenen Fehlerbalkenerstellung bleibt unberücksichtigt, dass ein Messwiederholungsfaktor dargestellt wird. Folglich harmoniert der optische Eindruck aus dem Vergleich der beiden Fehlerbalken oft schlecht mit der inferenzstatistischen Beurteilung durch den t-Test für abhängige Stichproben, der die interindividuellen Unterschiede aus den Fehlervarianzen eliminiert. Field (2012, S. 361ff) schlägt daher vor, aus den Messwerten die Personeneffekte zu entfernen und die Fehlerbalken aus diesen adjustierten Messwerten zu erstellen.

9.3.4.3.4 Gruppierte Balken

Bislang wurden einfache Balkendiagramme vorgestellt, die entweder eine univariate Verteilung oder den Effekt *eines* (gruppierten oder messwiederholten) Faktors auf eine abhängige Variable darstellen. Nun erstellen wir ein *gruppiertes* Balkendiagramm, das die kombinierten Effekte von zwei Faktoren auf ein metrisches Kriterium zeigt. Wie in Abschnitt 9.3.4.3.3 zu sehen war, werden Messwiederholungsfaktoren durch den Wechsel zum Langformat auf den Gruppierungsfall zurückgespielt. Im aktuellen Abschnitt kommt eine Langvariante der Datendatei **kfa.sav** zum Einsatz, die auch den Faktor `geschl` aus der Ausgangstabelle übernimmt (siehe **melt()** - Argument **id.vars**):

```
> library(reshape2)
> kfa.long <- melt(kfa, id.vars = c("fnr","geschl"),
+   measure.vars = c("aergo","aergm"), variable.name = "kfa")
> kfa.long$kfa <- factor(kfa.long$kfa, labels=c("Ohne","Mit"))
```

Um das folgende Ergebnis zu erzielen,



wird das ästhetische Attribut **fill** auf den Faktor `geschl` abgebildet:

```
> ggplot(kfa.long, aes(kfa,value,fill=geschl)) +
+   geom_bar(stat="summary",fun.y=mean,position=position_dodge(),colour="black") +
+   geom_errorbar(stat="summary", fun.data=mean_cl_normal,
+     position=position_dodge(width=0.9),width=0.5,size=1) +
+   labs(x="KFA", y="Mittlerer Ärger\n", fill="Geschlecht") +
+   scale_fill_manual(values=c("mediumorchid", "lightsteelblue3"))
```

An den beiden X-Achsenpositionen (definiert durch den Faktor `kfa`) sind jeweils *zwei* Balken auszugeben (für Frauen und Männer). Ohne Maßnahme zur Positionsanpassung (vgl. Abschnitt 9.3.1.5) würden die Balken übereinander gestapelt. Mit der Positionsanpassungsmethode **position_dodge()** wird stattdessen die Ausgabe von gruppierten Balken erreicht. Auf der Schicht mit den Fehlerbalken benötigt die Funktion **position_dodge()** das Argument **width**, weil Balken und Fehlerbalken unterschiedlich breit sind. Mit der Funktion **scale_fill_manual()** wird für eine individuelle Farbauswahl gesorgt.

Zusammen mit einer Rahmenfarbe für die Balken (z.B. `colour="black"`) erhält man leider auch eine diagonale Linie in jedem Legendenelement (siehe obige Abbildung). Mit dem folgenden Trick entfällt die Diagonale:

- Man erzeugt zwei **bar**-Schichten.
- Die erste Balkenschicht erhält *keine* Rahmenfarbe.
- Die zweite Balkenschicht erhält die gewünschte Rahmenfarbe, wird aber mit dem folgenden **geom_bar()** - Argument von der Legendenbildung ausgeschlossen:
  `show_guide=FALSE`

Leider verschwindet aber auch die Umrahmung der Legendenelemente:

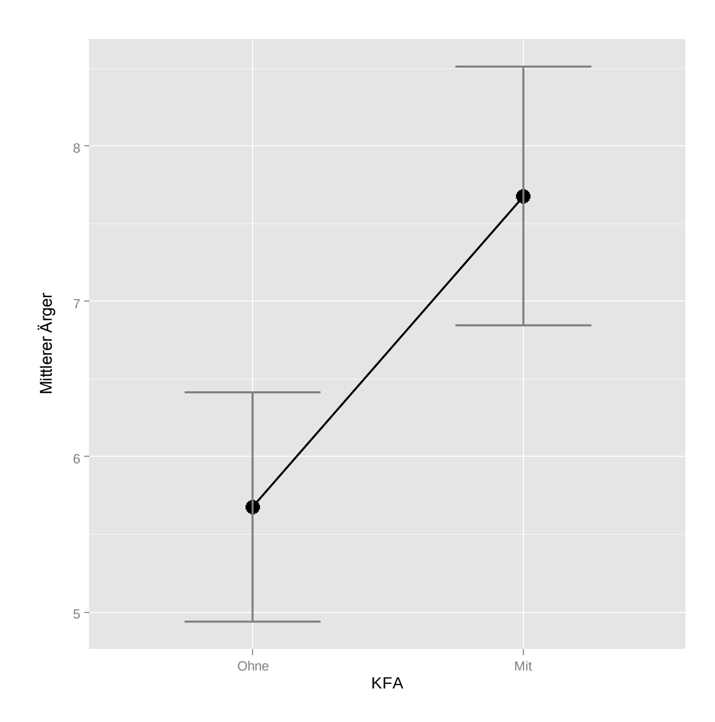


Im Beispiel führt der Trick zum folgenden Aufruf:

```
> ggplot(kfa.long,aes(kfa,value,fill=geschl)) +
+   geom_bar(stat="summary", fun.y=mean, position=position_dodge()) +
+   geom_bar(stat="summary", fun.y=mean, position=position_dodge(),
+     colour="black", show_guide=FALSE) +
+   . . .
```

9.3.4.4 Liniendiagramme

Wir erstellen zunächst eine Linienvariante des in Abschnitt 9.3.4.3.3 vorgestellten Balkendiagramms mit dem Messwiederholungsfaktor KFA:



Im Schichtenaufbau werden zunächst mit **geom_point()** zwei Punkte zur Darstellung der beiden Mittelwerte ausgegeben, wobei wie auf der Schicht mit dem Geom **line** die statistische Transformation **summary** mit der Zusammenfassungsfunktion **fun.y** zum Einsatz kommt:

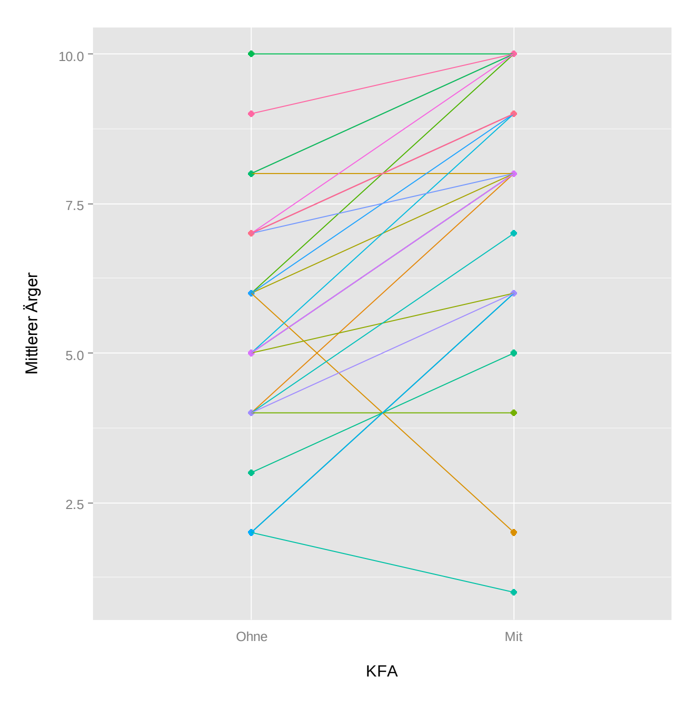
```
> ggplot(kfa.long,aes(x=kfa,y=value)) +
+   geom_point(stat="summary",fun.y=mean, size=5) +
+   geom_line(stat="summary", fun.y=mean, aes(group=1), size=1) +
+   geom_errorbar(stat="summary", fun.data=mean_cl_normal, width=0.5,
+     size=1,colour="gray50") +
+   labs(x="KFA", y="Mittlerer Ärger\n")
```

Im Aufruf von **geom_line()** ist die Abbildung des ästhetischen Attributs **group** auf den Wert 1 zu beachten. Damit wird die Gruppierung aufgrund des X-Achsen-Faktors KFA aufgehoben, und die beiden aus der Zusammenfassung hervorgegangenen Punkte gehören zu *einer* Gruppe. Diese Maßnahme ist erforderlich, weil das Geom **line** zu jeder Gruppe eine Linie erstellt, die alle enthaltenen Punkte verbindet.

Um die Veränderung zwischen den beiden Bedingung (ohne bzw. mit KFA) für jede einzelne Person zu betrachten, legen wir die Variable `fnr` mit der Fallnummer als farbdefinierend und gruppenbildend fest, nachdem sie in einen Faktor gewandelt worden ist:

```
> ggplot(kfa.long, aes(x=kfa,y=value,colour=factor(fnr))) +
+   geom_point(show_guide=FALSE) +  geom_line(aes(group=fnr), show_guide=FALSE) +
+   labs(x="\nKFA", y="Mittlerer Ärger\n")
```

Es resultiert ein „Spaghettidiagramm“, wobei die Legenden zum **point**- bzw. **line**-Geom durch den Wert **FALSE** für das Argument **show_guide** unterdrückt werden:



Vorschläge zu vielen weiteren Liniendiagrammen finden sich z.B. bei Chang (2013, S. 49ff) und Field (2012, S. 155ff).

9.3.5 ggplot2 - Diagramm in eine Datei sichern

Zum Sichern eines **ggplot2** - Diagramms in eine Datei steht neben den in Abschnitt 9.1 beschriebenen Optionen die **ggsave()** - Funktion zur Verfügung. Im ersten Parameter gibt man den Dateinamen an, wobei das Dateiformat (bzw. das zu verwendende Ausgabegerät) automatisch aus der Namenerweiterung abgeleitet wird, z.B.:

```
> ggsave("kfa.svg")
```

Man kann den Namen des zu sichernden Plots weglassen, wenn der zuletzt angezeigte Plot gemeint ist. Über die optionale Argumente **width** und **height** lassen sich Breite und Höhe in der Einheit Zoll (engl.: *Inch*) festlegen (1 Zoll = 2,54 cm), z.B.:

```
> ggsave("kfa.svg", width=15, height=15)
```

Bei einem Bitmap-Format (z.B. PNG) kann die Auflösung über das Argument **dpi** gewählt werden, z.B.:

```
> ggsave("kfa.png", width=15, height=15, dpi=600)
```

Ein Bitmap-Format mit einer Auflösung von 600 dpi ist z.B. dann zu empfehlen, wenn ein Diagramm unter Windows an ein Textverarbeitungsprogramm (Libre-, MS- oder Open-Office) übergeben werden soll (siehe Wickham 2009, Abschnitt 8.3).

Das unter Windows populäre Metafile- bzw. Vektordateiformat WMF bzw. EMF ist zur Aufnahme von **ggplot2**-Grafiken *nicht* zu empfehlen:

- Es unterstützt keine Transparenz, so dass z.B. die von **geom_smooth()** erstellten Konfidenzintervalle (siehe Abschnitt 9.3.2) verloren gehen.
- Im Vergleich zu anderen Vektorformaten (z.B. SVG) sind Kurven sehr grob aufgelöst.

Das traditionelle **R**-Verfahren zur Grafikausgabe in eine Datei ist z.B. dann gegenüber der Funktion **ggsave()** zu bevorzugen, wenn mehrere Diagramme jeweils auf einzelnen Seiten einer PDF-Datei abgelegt werden sollen.

10 Weitere Anwendungen von R

10.1 Mengenlehre

Ein Vektor kann eine Menge repräsentieren. Dann ...

- spielt die Reihenfolge der Elemente keine Rolle,
- werden Dubletten ignoriert.

Um Dubletten explizit zu entfernen, verwendet man die Funktion **unique()**, z.B.:

```
> set1 <- c(1,2,2,3,4,5)
> (uset <- unique(set1))
[1] 1 2 3 4 5
> length(unique(set1))
[1] 5
```

Den Durchschnitt zweier Mengen liefert die Funktion **intersect()**, z.B.:

```
> set2 <- c(3,4,5,6,7)
> intersect(set1, set2)
[1] 3 4 5
```

Um zwei Mengen zu vereinigen, verwendet man die Funktion **union()**, z.B.:

```
> union(set1, set2)
[1] 1 2 3 4 5 6 7
```

Subtrahiert man über die Funktion **setdiff()** von einer Menge `set1` eine Menge `set2`, dann resultiert als Differenz eine Menge mit allen `set1`-Elementen, die sich *nicht* in `set2` befinden, z.B.:

```
> setdiff(set1, set2)
[1] 1 2
```

Ob sich ein bestimmtes Element in einer Menge befindet, prüft man mit der Funktion **is.element()**, z.B.:

```
> is.element(2, set1)
[1] TRUE
```

Wird für einen Vektor die elementweise Existenzprüfung vorgenommen, resultiert ein Vektor vom Typ **logical**, z.B.:

```
> is.element(c(0,1,2), set1)
[1] FALSE  TRUE  TRUE
```

Alternativ zur Funktion **is.element()** kann mit demselben Ergebnis der Operator **%in%** verwendet werden, z.B.:

```
> c(0,1,2) %in% set1
[1] FALSE  TRUE  TRUE
```

Um für eine Menge zu prüfen, ob sie **Teilmenge** einer anderen Menge ist, wendet man die Funktion **all()** auf die Rückgabe der Funktion **is.element()** bzw. des Operators **%in%** an, z.B.:

```
> all(c(0,1,2) %in% c(0:4))
[1] TRUE
```

Weitere Mengenoperationen mit **R** werden in Wollschläger (2010, S. 40ff) beschrieben.

10.2 Lineare Algebra

Um das folgende Gleichungssystem zu lösen,

$$\begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

kann man die **R**-Funktion **solve()** verwenden, die als ersten Parameter die Koeffizientenmatrix und als zweiten Parameter den Zielvektor (oder die Zielmatrix) erwartet.

```
> xmat <- matrix(c(1, 2, 2, 3), 2)
> xmat
     [,1] [,2]
[1,]    1    2
[2,]    2    3
> b <- c(5, 8)
> solve(xmat, b)
[1] 1 2
```

Lässt man den zweiten Parameter weg, wird dort die Einheitsmatrix angenommen, so dass die Inverse der Koeffizientenmatrix als Lösung verlangt wird:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

Im Beispiel:

```
> ix <- solve(xmat)
> ix
     [,1] [,2]
[1,]   -3    2
[2,]    2   -1
```

Um das Ergebnis zu prüfen, multiplizieren wir die Matrizen `xmat` und `ix`, wobei das Operatorzeichen **`%*%`** zu verwenden ist:

```
> xmat %*% ix
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Literatur

Baltes-Götz, B. (2014a). *Lineare Regressionsanalyse mit SPSS*. Online-Dokument: http://www.uni-trier.de/index.php?id=22489

Baltes-Götz, B. (2014b). *Mediator- und Moderatoranalyse per multipler Regression mit SPSS*. Online-Dokument: http://www.uni-trier.de/index.php?id=22528

Baltes-Götz, B. (2014c). *Statistisches Praktikum mit SPSS für Windows*. Online-Dokument: http://www.uni-trier.de/index.php?id=22552

Bellio, R. & Ventura, L. (2005). *An Introduction to Robust Estimation with R Functions*. Online-Dokument (abgerufen: 17.11.2013): http://www.dst.unive.it/rsr/BelVenTutorial.pdf

Bliese, P.D. (2013). *Multilevel Modeling in R 2.5). A Brief Introduction to R, the multilevel package and the nlme package*. Online-Dokument (abgerufen: 07.11.2014): http://cran.r-project.org/doc/contrib/Bliese_Multilevel.pdf

Chambers, J. (1998). *Programming with Data. A Guide to the S Language*. New York: Springer.

Chang, W. (2013). *R Graphics Cookbook* (2nd ed.). Beijing: O'Reilly.

Cohen , R. (2013). *Extension Bundles from IBM SPSS*. Online-Dokument (abgerufen: 30.11.2014): https://www.ibm.com/developerworks/community/files/form/anonymous/api/library/b5bb8a42-04d2-4503-93bb-dc45d7a145c2/document/1d445b76-d706-44a6-85bf-9e3738d223a4/media/Extension%20Bundles%20from%20IBM%20SPSS.pdf

Field, A. (2012). *Discovering Statistics Using R*. London: SAGE Publications.

Fox, J. (2002). *Robust Regression*. Online-Dokument (abgerufen am 21.01.2011): http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-robust-regression.pdf.

Fox, J. (2005). The R Commander: A Basic Statistics Graphical User Interface to R. *Journal of Statistical Software*, 14(9), 1-42.

Fox, J. & Weisberg, S. (2011). *An R Companion to Applied Regression* (2nd ed). Thousand Oaks: SAGE Publications.

Fox, J. & Bouchet-Valat, M. (2013). *Getting Started With the R Commander*. Online-Dokument (abgerufen: 16.08.2014): http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/Getting-Started-with-the-Rcmdr.pdf

Gordon, R.A. (2010). *Regression Analysis for the Social Sciences*. New York: Routledge.

Hain, J. (2011). *Statistik mit R*. RRZN-Handbuch.

IBM SPSS (2013). *IBM SPSS Statistics 22 Core-System Benutzerhandbuch*. Online-Dokument: ftp://public.dhe.ibm.com/software/analytics/spss/documentation/statistics/22.0/de/client/Manuals/IBM_SPSS_Statistics_Core_System_User_Guide.pdf

Levesque, R & IBM Inc. (2011). *Programming and Data Management for SPSS Statistics 20*. IBM Inc. Online-Dokument (abgerufen am 23.09.2011): https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/We70df3195ec8_4f95_9773_42e448fa9029/page/Books%20and%20Articles

Ligges, U. (2007). *Programmieren mit R* (2. Aufl.). Berlin: Springer.

Muenchen, R.A. (2011). *R for SAS and SPSS Users* (2. Aufl.). New York: Springer.

Murrell, P. (2011). *R Graphics* (2nd. ed.). Boca Raton, FL: Chapman & Hall/CRC.

R Development Core Team (2014). *R Language Definition*. Online-Dokument (abgerufen am 30.11.2014): http://cran.r-project.org/doc/manuals/R-lang.pdf

Ryan, T.S. (1997). *Modern Regression Methods*. New York: Wiley.

IDRE UCLA (2014a). *R Data Analysis Examples: Robust Regression*. Online-Dokument (abgerufen am 30.11.2014): http://www.ats.ucla.edu/stat/r/dae/rreg.htm

IDRE UCLA (2014b). *R Data Analysis Examples. Tobit Models*. Online-Dokument (abgerufen am 30.11.2014): http://www.ats.ucla.edu/stat/R/dae/tobit.htm

Venables, W. N., Smith, D. M. & R Core Team. (2014). *An Introduction to R*. Online-Dokument (abgerufen am 30.11.2014): http://cran.r-project.org/doc/manuals/r-devel/R-intro.pdf.

Weisberg, S. (1985). *Applied linear regression* (2nd ed.). New York: Wiley.

Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. New York: Springer.

Wilkinson, L. (2005). *The Grammar of Graphics* (2nd ed.). New York: Springer.

Wollschläger, D. (2010). *Grundlagen der Datenanalyse mit R*. Heidelberg: Springer.

Stichwortverzeichnis

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T